UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Aplicación para la gestión de pistas de pádel

Autor/es

SERGIO PÉREZ VEGA

Director/es

BEATRIZ PÉREZ VALLE

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Ingeniería Informática

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2016-17

Aplicación para la gestión de pistas de pádel, de SERGIO PÉREZ VEGA (publicada por la Universidad de La Rioja) se difunde bajo una Licencia Creative Commons Reconocimiento-NoComercial-SinObraDerivada 3.0 Unported. Permisos que vayan más allá de lo cubierto por esta licencia pueden solicitarse a los titulares del copyright.

© El autor, 2017
© Universidad de La Rioja, 2017
publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Aplicación para la gestión de pistas de pádel

Alumno:

Sergio Pérez Vega

Tutores:

Beatriz Pérez Valle

Logroño, Junio, 2017

Resumen

Este proyecto consiste en crear una aplicación multiplataforma para que los usuarios puedan reservar pistas de pádel de un club de pádel ficticio, en el momento que deseen y desde donde deseen de una forma fácil y sencilla.
La aplicación va a estar dividida en dos partes. Una parte dirigida al administrador de la aplicación para que pueda gestionar todos los elementos del contexto, que van a ser principalmente pistas, usuarios y reservas. La otra parte de la aplicación va dirigida a los usuarios de la misma, mediante la cual podrán reservar pistas de pádel desde sus móviles o mediante un ordenador de mesa, a través de un navegador web.

En este proyecto partiremos de la premisa de que cada partido de pádel se juega con cuatro jugadores, lo cual a veces no es posible. Esto hace que se reduzca el número de reservas en los clubes de pádel. Por eso, la aplicación que se presenta va a tener la posibilidad de reservar una pista de pádel sin contar con cuatro jugadores, para que posteriormente otros usuarios se puedan unir al partido y así sea más fácil completarlo con cuatro jugadores.

Abstract

This project consists of creating a cross-platform application so that users can reserve paddle courts of a fictional paddle club, whenever and wherever they want, in an easy and simple way.

The application will be divided into two parts. A part addressed to the administrator of the application so that it can manage all the elements of the context, which are going to be mainly courts, users and reservations. The other part of the application is addressed to the users of the same, by means of which they can reserve paddle courts from their mobile phones or through a desktop computer, using a web browser.

In this project we will start from the premise that each paddle game is played with four players, which sometimes is not possible. This reduces the number of reserves in paddle clubs. Therefore, the application that is presented will have the possibility of reserving a paddle court without having four players, so that later, other users can join the game and make it easier to complete with four players.

Contenido

1. Introducción

El presente Trabajo Fin de Grado (TFG) va a consistir en la realización de una aplicación multiplataforma que tiene como objetivo facilitar la gestión de las pistas de pádel de un club de pádel ficticio. En particular, se pretende que esta aplicación permita realizar la reserva de las pistas de una forma fácil e intuitiva.

Más explícitamente, los usuarios podrán reservar las pistas sin contar con cuatro personas para jugar[1]. La idea es que un usuario pueda reservar una pista de pádel en un día y una hora en concreto sin disponer de cuatro jugadores para jugar el partido y que la aplicación permita a los demás usuarios unirse a ese partido hasta que haya cuatro jugadores para jugarlo. Los jugadores estarán clasificados en la aplicación por el nivel de juego que tienen a la hora de jugar a pádel, de forma que puedan jugar contra jugadores de su mismo nivel.

El administrador podrá gestionar todos los elementos involucrados en la aplicación (reserva de pistas, usuarios, etc.). Asimismo, podrá consultar las estadísticas de uso de las diferentes pistas gestionadas por la aplicación.

El presente trabajo se ha desarrollado en la empresa Netbrain Media Solutions S.L., donde Hernán González, director de proyectos de esta empresa, actuará de cliente.

2. Motivación

Este TFG nace del interés de abordar las diferentes funcionalidades citadas anteriormente a través de una aplicación que pueda ser utilizada en el futuro por algún club de pádel. En particular, para favorecer un mayor uso de las pistas de pádel, se considera que algunos clubes pueden estar interesados en una aplicación móvil que permita gestionar la reserva de pistas de forma online, además de una aplicación web. Por ello este proyecto plantea el desarrollo de ambas aplicaciones.

[1] En este proyecto, y a petición del cliente, partiremos de la premisa de que cada partido de pádel se juega con cuatro jugadores

3. Planificación

A continuación se muestra la planificación que se ha realizado para lleva a cabo el presente proyecto. El total de las horas a invertir en el proyecto serán 300 horas, repartidas según se indica en la siguiente tabla:

Fase	Horas	Inicio	Fin	Explicación
1. Planificación	**6**	**07/02/17**	**08/02/17**	**Planificar las tareas a realizar dentro del proyecto.**
2. Estudio de la tecnología	**20**	**02/02/17**	**09/02/17**	**Estudiar y aprender a usar la tecnología a utilizar para el desarrollo del proyecto.**
3. Análisis de requisitos	**29**	**09/02/17**	**22/02/17**	**Analizar y definir los requisitos necesarios para realizar el proyecto.**
3.1. Reuniones con el cliente	4	09/02/17	09/02/17	Acordar con el cliente los requisitos necesarios de la aplicación.
3.2. Determinar los requisitos a incluir en el proyecto	25	13/02/17	22/02/17	Estudiar lo acordado con el cliente y decidir el alcance del proyecto.
4. Diseño	**45**	**22/02/17**	**13/03/17**	**Diseñar la funcionalidad y los prototipos de las interfaces de la aplicación.**
4.1. Diseño de la base de datos	25	22/02/17	02/03/17	Diseño de la base de datos.
4.2. Diseño de la interfaz	20	06/03/17	13/03/17	Diseño de los prototipos de las interfaces que va a tener la aplicación.
5. Implementación	**145**	**13/03/17**	**25/05/17**	**Implementación de la aplicación**
5.1. Creación de la base de datos	30	13/03/17	23/03/17	Implementación de la base de datos a utilizar.
5.2. Creación de las interfaces de la aplicación	40	27/03/17	11/04/17	Implementación de la interfaz de usuario.
5.3. Implementar la funcionalidad de la aplicación	75	11/04/17	25/05/17	Implementar la funcionalidad necesaria de la aplicación.
6. Pruebas	**10**	**29/05/17**	**01/06/17**	**Realizar pruebas de la aplicación.**
7. Redactar memoria	**30**	**02/02/17**	**02/06/17**	**Redactar la memoria del proyecto.**
7.1. Diario de lo sucedido	9	02/02/17	02/06/17	Usar media hora cada semana para recoger lo realizado en esa semana.
7.2. Redacción de la memoria	11	29/05/17	31/05/17	Redactar la memoria con todo lo recogido en los pasos anteriores.
7.3. Preparación de la presentación del proyecto	10	31/05/17	02/06/17	Realización y estudio de la presentación para la defensa del proyecto.
8. Reuniones con la tutora de la Universidad	**15**	**02/02/17**	**02/06/17**	**Realizar reuniones periódicas con la tutora de la Universidad.**

4. Tecnologías a utilizar

Las tecnologías usadas para la realización de este proyecto son las siguientes:

- *Angular 2* (también llamado *AngularJS)*: es un framework de *Javascript* de código abierto, mantenido por Google. Su objetivo es favorecer el uso del patrón *MVC* (Modelo Vista Controlador) para aplicaciones web, que sirve para hacer que el desarrollo sea más fácil y reducir la manipulación del *DOM* (Document Object Model refiere a un conjunto estándar de objetos para representar documentos *HTML*). Este framework completa el *HTML* para añadir contenido dinámico que permite la sincronización bidireccional automática entre modelos y vistas.

  Hemos elegido esta tecnología porque facilita la bidireccionalidad entre los modelos y las vistas y porque es un framework que está siendo muy utilizado para la realización de aplicaciones web. Ha sido usado principalmente para la comunicación entre la capa de presentación y la capa de lógica de negocio [3,4].
- *Ionic 2*: construido en *AngularJS* y *Apache Cordova* proporciona herramientas y servicios para el desarrollo de aplicaciones móviles híbridas usando tecnologías web como *HTML5* y *CSS*. Es el framework utilizado para el desarrollo y la compilación del proyecto en las diferentes plataformas como *Android, IOS* o *Windows*.

  Hemos elegido esta tecnología porque nos permite escribir el código de la aplicación una sola vez y compilar ese mismo código en diferentes plataformas. Además, siempre te permite añadir nuevas plataformas así se requiera. Otro motivo para su elección es que utiliza *AngularJS*, que coincidimos en que era una tecnología muy utilizada actualmente y que puede ser de gran ayuda en un futuro. Aunque *Ionic 2* tiene la pega de que, al no tratarse de una tecnología nativa, en un principio es una plataforma menos eficiente en cuanto a la velocidad de las aplicaciones, se ha decidido darle menos importancia a este aspecto debido a que nuestra aplicación no va a tener una gran carga gráfica en los que sí se podría notar [1,2].
- *Node JS*: es un entorno en tiempo de ejecución multiplataforma, de código abierto basado en el motor V8 de Google. Usa un modelo con un único hilo de ejecución con operaciones E/S sin bloqueo y orientado a eventos. Está diseñado para construir aplicaciones en red escalables. Se ha usado para la instalación de *Ionic 2* [7].
- *Typescript*: es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft. Es un superconjunto de *JavaScript* que añade tipado estático y objetos basados en clases. A través de un compilador de *Typescript* se traduce el código *Typescript* en código *Javascript*. Está pensado para grandes proyectos. Se ha elegido esta tecnología porque es la principal tecnología utilizada en proyectos con *Ionic2*. En particular, se ha usado para la implementación de la capa de lógica de negocio [5,6].

- *Firebase*: es una plataforma de desarrollo para aplicaciones móviles y web que utiliza la infraestructura de *Google*. Se compone de características que los desarrolladores pueden mezclar y combinar para ajustarlo a sus necesidades. El producto inicial de *Firebase* era una base de datos en tiempo real que permitía a los desarrolladores almacenar y sincronizar los datos a través de múltiples clientes. Con el tiempo se ha ampliado el conjunto de productos que ofrece para convertirse en un producto completo para desarrolladores. De los diferentes productos que ofrece *Firebase*, en este proyecto se van a usar:
    - *Realtime Database*: proporciona una base de datos en tiempo real que almacena los datos en una base de datos *NoSQL*. Los datos son almacenados en archivos con formato *JSON*. También proporciona herramientas para el acceso y la modificación de los datos en la base de datos. Usada para almacenar los datos necesarios para el proyecto.
    - *Firebase Auth*: proporciona diferentes herramientas para controlar la autenticación de los usuarios dentro de una aplicación. Para ello nos proporciona diferentes posibilidades de autenticación. En particular, en este proyecto se utilizará la autenticación mediante email y contraseña.

  El motivo por el cual se ha elegido esta tecnología, es principalmente porque es un servicio de *Google*, lo que da bastantes garantía. Además es un servicio ya probado y, en nuestra opinión, más fiable que uno realizado por nosotros mismos. También nos ahorraba el tener que crear nuestro propio servicio. Es altamente accesible y nos permite diferentes servicios para obtener determinada funcionalidad requerida en el proyecto, como por ejemplo, el servicio de autenticación. Por último, es fácilmente escalable en caso de que nuestra aplicación siga creciendo, porque de esta parte se encarga *Google* [16,17].
- *Visual Studio Code*: es un entorno de desarrollo integrado para sistemas operativos Windows. Este entorno soporta múltiples lenguajes de programación. En particular, es este proyecto se ha utilizado para desarrollar con *Ionic 2, AngularJS, Typescript, HTML5* y *CSS* [11].
- *Android Studio*: entorno de desarrollo integrado oficial para el desarrollo de aplicaciones para Android basado en *IntelliJ IDEA*. Tiene un sistema de compilación basado en *Gradle* y permite emular las aplicaciones desarrolladas en este entorno desde dispositivos móviles conectados al ordenador por *USB* sin necesidad de compilar un nuevo *APK* (es el archivo ejecutable de las aplicaciones en sistemas *Android*). Usado para que, una vez terminado el proyecto en *Ionic 2* y compilado para la plataforma *Android*, podamos ejecutar el archivo *APK* en un dispositivo android [8].
- *Github*: es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones *Git*. Permite almacenar el código de forma pública aunque también se puede hacer de forma privada creando una cuenta de pago. En nuestro caso se ha utilizado para alojar el proyecto y para controlar las versiones del mismo. En particular, se ha usado Github para alojar el desarrollo de este proyecto [13].

- *Balsamiq mockups*: es una aplicación que permite diseñar prototipos de interfaces. Usada para el diseño de los prototipos de las interfaces del proyecto creados en la fase de diseño [14].
- *ArgoUML*: es una aplicación que permite diseñar diagramas *UML*, en particular los diagramas de Casos de Uso y Diagramas de clases creados en la fase de análisis [15].
- Además de todas las tecnologías anteriores también se ha utilizado *CSS* y *HTML5* [9,10].

En este punto merece la pena destacar el gran número de tecnologías utilizadas para la realización de estre proyecto, la mayoría de las cuales nuevas para el alumno.

5. Plan de contingencia

Los principales riesgos que podrían tener lugar durante el desarrollo del presente proyecto, su nivel de impacto y la forma de intentar corregirlos son los siguientes:

Riesgo: Cambio de requisitos.

- *Impacto*: Dependerá de la fase en la que se encuentre el proyecto. Cuanto más avanzado esté el proyecto mayor impacto negativo tendrá.
- *Nivel de probabilidad de que ocurra*: Medio.
- *Fases en las que se puede producir*: Todas las fases.
- *Plan de contingencia*: Dejar bien claro en el análisis de requisitos el alcance del proyecto. En el caso de que el cliente sugiera alguna modificación, estudiar la posibilidad de llevarla a cabo o no.

Riesgo: Cambios de la tecnología o problemas con la misma.

- *Impacto*: Muy alto.
- *Nivel de probabilidad de que ocurra*: Alto.
- *Fases en las que se puede producir*: Fase de implementación.
- *Plan de contingencia*: Establecer antes de empezar la implementación cuál es la mejor tecnología a utilizar, una vez elegida, estudiarla. En el caso de descubrir que no se consigue avanzar con la tecnología escogida, dependerá del momento en el que nos encontremos. Si no estamos en un momento avanzado de la implementación, se elegirá otra tecnología. Pero si estamos en un momento avanzado de la implementación, se llegará a un consenso con el cliente para intentar suplir los problemas.

Riesgo: Problemas derivados de una mala planificación.

- *Impacto*: Alto.
- *Nivel de probabilidad de que ocurra*: Medio.
- *Fases en las que se puede producir*: Todas las fases.
- *Plan de contingencia*: Crear una planificación lo más realista posible. En caso de no poder cumplir la planificación, procurar reajustar la planificación en el momento en el que nos demos cuenta.

Riesgo: Problemas personales que impiden realizar el trabajo en las horas asignadas para realizarlo.

- *Impacto*: Alto.
- *Nivel de probabilidad de que ocurra*: Medio.
- *Fases en las que se puede producir:* Todas las fases.
- *Plan de contingencia*: Intentar que no ocurra y, en el caso de que dichos problemas tengan lugar, conseguir tiempo no incluido en la planificación (como por ejemplo algún fin de semana) con objeto de recuperar dichas horas.

Riesgo: Pérdida de la documentación.

- *Impacto*: Muy alto.
- *Nivel de probabilidad de que ocurra*: Bajo.
- *Fases en las que se puede producir*: Todas las fases.
- *Plan de contingencia*: Creación de copias de seguridad de toda la información correspondiente al proyecto en diferentes dispositivos. En nuestro caso se utilizará un disco duro externo, un *USB* y *Google Drive*. Además, como se ha comentado anteriormente, se almacenará la información en *GitHub*.

6. Análisis

En este capítulo se describe el análisis de requisitos, el diagrama de casos de uso y la especificación de los casos de uso.

6.1. Análisis de requisitos

A continuación describiremos el análisis de requisitos de la aplicación, los cuales han sido establecidos por el cliente. Se distingue entre requisitos funcionales y requisitos no funcionales.

6.1.1. Requisitos funcionales

Dividiremos estos requisitos en función de si refieren a la parte gestionada por el administrador o a la parte correspondiente a los usuarios (tanto visitantes como registrados). Ambas partes dispondrán de una versión móvil y web.

En la aplicación para el administrador en ambas versiones (versión móvil y versión web):

- El administrador podrá autenticarse para acceder a la aplicación.
- El administrador podrá ver la disponibilidad de las pistas de pádel sin reservar. Desde aquí podrá seleccionar una pista a una hora y un día en concreto para reservarla en nombre de un usuario.
- El administrador podrá gestionar los usuarios creando nuevos usuarios y viendo la lista de los ya creados. Desde la lista de usuarios podrá modificar los datos personales del usuario que desee y también podrá eliminarlo.
- El administrador podrá gestionar las pistas del club de pádel creando pistas nuevas y viendo la lista de las ya creadas. Desde la lista de las pistas podrá eliminar la pista que desee.
- El administrador podrá gestionar las reservas de las pistas realizadas por los usuarios dentro de la aplicación, desde donde verá la lista de las reservas tanto activas como las ya ejecutadas. Desde ahí, podrá eliminar la reserva que desee.
- El administrador podrá ver estadísticas sobre el uso de las pistas y sobre las reservas.

En la aplicación para los usuarios visitantes en ambas versiones:

- Todo usuario visitante, si así lo desea, se podrá registrar en la aplicación para poder usarla proporcionando como datos el nombre, el DNI, el teléfono, el nivel de juego, un nombre de usuario, la contraseña, si es abonado del club de pádel o no y el email.

Respecto a los usuarios registrados, distinguiremos entre la versión web y la versión móvil:

- En la versión web:
    - El usuario podrá autenticarse para acceder a la aplicación en el momento que desee.
    - El usuario podrá modificar sus datos personales en cualquier momento.
    - El usuario podrá ver en todo momento la disponibilidad de las pistas sin reservar. Desde aquí, el usuario podrá reservar una pista en el momento que desee siempre que la pista no esté ya reservada.
    - El usuario podrá ver el historial de reservas que ha realizado.
    - El usuario podrá elegir si quiere recibir notificaciones o no.

- En la versión móvil, los usuarios registrados tendrán los mismos requisitos que en la versión web, añadiendo a estos requisitos el siguiente:
    - Cuando un usuario registrado reserve una pista con un número de jugadores inferior a cuatro, el sistema enviará una notificación a todos los usuarios de su mismo nivel y que tengan activas las notificaciones, de forma que estos puedan unirse al partido.

Nota: La idea inicial era realizar el envío de notificaciones en la versión móvil de los usuarios registrados y por lo tanto los siguientes pasos del análisis y diseño incluyen este requisito. Pero, como ya explicaré más adelante, este requisito no se ha podido implementar debido a dificultades con la tecnología.

6.1.2. Requisitos no funcionales

En este apartado se presentan los requisitos no funcionales:

- La interfaz tanto web como móvil será fácil e intuitiva con objeto de facilitar el uso a los usuarios y al administrador.
- La aplicación móvil deberá funcionar en Android versión 6.0.1 y la aplicación web funcionará en los navegadores *Google Chrome* versión 58.0.3 y *Mozilla Firefox* versión 53.0.3.

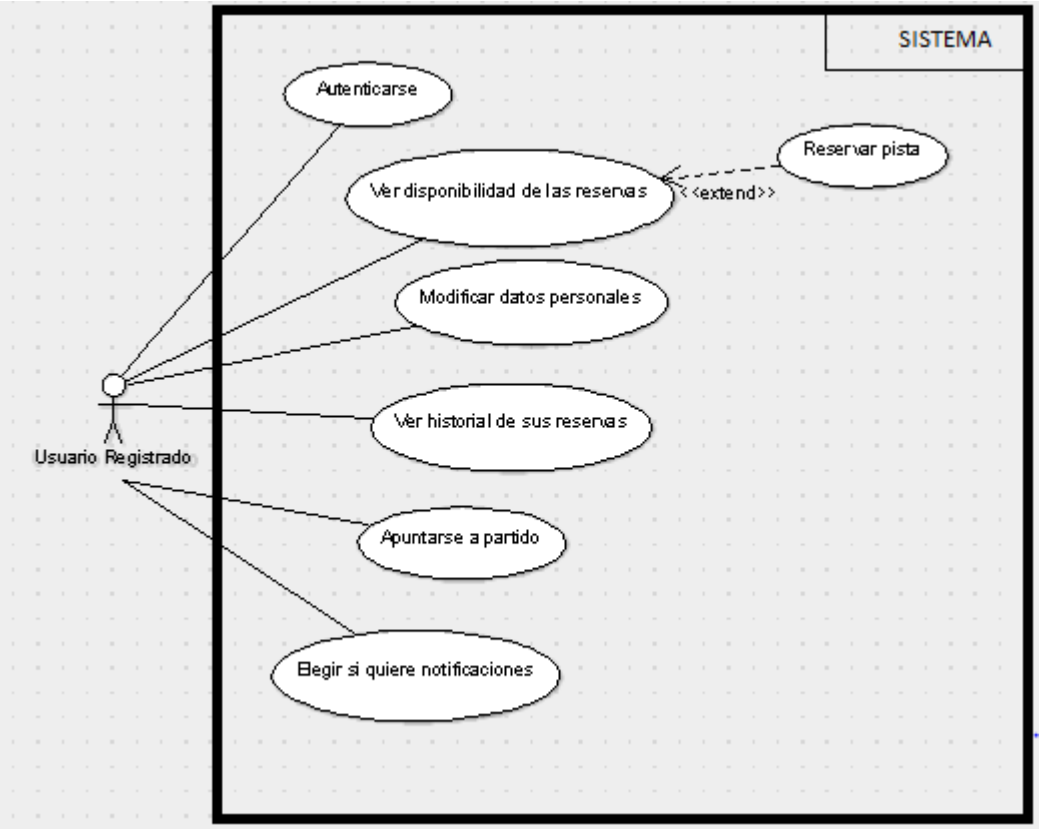
6.2. Diagrama de casos de uso

En base a los anteriores requisitos se han identificado tres actores que interactúan con la aplicación que van a ser *Administrador*, *Usuario visitante* y *Usuario registrado*. A continuación se presentan los diagramas de casos de uso diseñados para cada caso.

El diagrama de casos de uso de la aplicación tanto web como móvil para el administrador es el siguiente:

El diagrama de casos de uso del actor *Usuario visitante* solamente contiene un caso de uso que es el de *Registrarse*.

En el diagrama de casos de uso del actor *Usuario registrado* sería el que se muestra a continuación:



En este diagrama se muestran los casos de uso del *usuario registrado* de la versión móvil. En el caso de la versión web, se tendrían los mismos casos de uso, pero quitando el caso de uso de *Apuntarse a partido*.

Notaremos que tanto el usuario *Administrador* como el *Usuario registrado* deberán estar registrados en la aplicación para realizar los casos de uso propuestos.

6.2.1. Especificación de los casos de uso

A continuación, se explica cada uno de los casos de uso presentados en el apartado anterior:

Caso de Uso: Autenticarse.
Objetivo: Iniciar sesión en la aplicación con su nombre de usuario y contraseña.
Actor(es): Administrador y Usuario registrado.
Precondición: El administrador o el usuario registrado deben estar dados de alta en la aplicación.
Pasos: 1. El administrador o usuario registrado accede a la aplicación. 2. Introduce el nombre de usuario y contraseña. 3. Accede a la aplicación.
Extensiones: 2.1. Introduce un nombre de usuario que no exista ya en la aplicación o una contraseña no válida. 3.1. El sistema le avisa de que el usuario o la contraseña introducidos no son correctos.

Caso de Uso: Ver disponibilidad de las pistas.
Objetivo: Ver la disponibilidad de las pistas sin reservar en los días deseados.
Actor(es): Administrador o el usuario registrado.
Precondición: El administrador o el usuario registrado deben haberse autenticado.
Pasos: 1. El administrador o el usuario registrado accede a la sección para *ver la disponibilidad de las pistas*. 2. Ve la disponibilidad de las pistas en un día en concreto.
Extensiones:

Caso de Uso: Reservar pista.
Objetivo: Reservar una pista para un usuario.
Actor(es): Administrador.
Precondición: El administrador debe haberse autenticado y la pista no puede estar reservada en el día y la hora deseada.
Pasos: 1. El administrador accede a la sección de ver *la disponibilidad de las pistas*. 2. Elige la hora, el día y la pista a reservar. 3. Introduce el nombre del usuario que la quiere reservar. 4. Reserva la pista.
Extensiones: 1.1. Introduce un nombre de usuario que no exista en la aplicación. 1.1.1. El sistema le avisa de que el nombre de usuario introducido no existe.

Caso de Uso: Gestión usuarios.
Objetivo: Acceder a la sección de gestión de usuarios.
Actor(es): Administrador.
Precondición: El administrador debe haberse autenticado.
Pasos: 1. El administrador accede a la sección de *gestión de usuarios*.
Extensiones:

Caso de Uso: Añadir usuario.
Objetivo: Añadir un usuario nuevo a la aplicación para que pueda utilizarla con su nombre de usuario y contraseña.
Actor(es): Administrador.
Precondición: El administrador debe haberse autenticado.
Pasos: 2. El administrador accede a la sección de *gestión de usuarios*. 3. Pulsa en el botón de añadir un usuario. 4. Añade los datos del nuevo usuario.
Extensiones:

Caso de Uso: Ver lista usuarios.
Objetivo: Ver todos los usuarios que hay dados de alta en la aplicación.
Actor(es): Administrador.
Precondición: El administrador debe haberse autenticado.
Pasos: 1. El administrador accede a la sección de *gestión de usuarios*. 2. Ve la lista de usuarios dados de alta.
Extensiones:

Caso de Uso: Modificar datos de usuario.
Objetivo: Modificar los datos de un usuario ya dado de alta en la aplicación.
Actor(es): Administrador.
Precondición: El administrador debe haberse autenticado y que el usuario que se va a modificar ya exista.
Pasos: 1. El administrador accede a la sección de *gestión de usuarios*. 2. Ve la lista usuarios. 3. Elige el usuario deseado. 4. Modifica los datos del usuario elegido. 5. Acepta cambios.
Extensiones:

Caso de Uso: Eliminar usuario.
Objetivo: Se eliminará un usuario de la aplicación.
Actor(es): Administrador.
Precondición: El administrador debe haberse autenticado.
Pasos: 1. El administrador accede a la sección de *gestión de usuarios*. 2. Ve la lista de usuarios. 3. Elige usuario deseado. 4. Elimina el usuario deseado.
Extensiones:

Caso de Uso: Gestión pistas.
Objetivo: Acceder a la sección de gestión de pistas.
Actor(es): Administrador.
Precondición: El administrador debe haberse autenticado.
Pasos: 1. El administrador accede a la sección de *gestión de pistas*.
Extensiones:

Caso de Uso: Añadir pista.
Objetivo: Añadir una pista nueva a la aplicación para que pueda ser reservada y utilizada.
Actor(es): Administrador.
Precondición: El administrador debe haberse autenticado.
Pasos: 1. El administrador accede a la sección de *gestión de pistas*. 2. Pulsa en el botón de añadir una pista. 3. Crea una pista nueva.
Extensiones:

Caso de Uso: Ver lista pistas.
Objetivo: Ver el listado de las pistas de pádel.
Actor(es): Administrador.
Precondición: El administrador debe haberse autenticado.
Pasos: 1. El administrador accede a la sección de *gestión de pistas*. 2. Ve la lista de pistas de pádel.
Extensiones:

Caso de Uso: Eliminar pista.
Objetivo: Eliminar de la aplicación una pista ya existente.
Actor(es): Administrador.
Precondición: El administrador debe haberse autenticado y que la pista esté dada de alta en la aplicación
Pasos: 1. El administrador accede a la sección de *gestión de pistas*. 2. Ve la lista de pistas. 3. Elige la pista deseada. 4. Elimina la pista deseada.
Extensiones:

Caso de Uso: Gestión reservas.
Objetivo: Acceder a la gestión de reservas.
Actor(es): Administrador.
Precondición: El administrador debe haberse autenticado y que la pista esté reservada a la fecha y hora indicada.
Pasos: 1. El administrador accede a la sección de *gestión de reservas* desde donde puede ver una lista de las reservas ya realizadas en la aplicación.
Extensiones:

Caso de Uso: Eliminar reserva pista.
Objetivo: Eliminar una reserva ya realizada.
Actor(es): Administrador.
Precondición: El administrador debe haberse autenticado y que la pista esté reservada a la fecha y hora indicada.
Pasos: 1. El administrador accede a la sección de *gestión de reservas*. 2. Elige la reserva deseada. 3. Elimina la reserva de la pista.
Extensiones:

Caso de Uso: Ver estadísticas de pistas.
Objetivo: Ver estadísticas sobre el uso de las pistas de pádel.
Actor(es): Administrador.
Precondición: El administrador debe haberse autenticado.
Pasos: 1. El administrador accede a las *estadísticas*. 2. Ve las estadísticas de las pistas.
Extensiones:

Caso de Uso: Ver estadísticas de reservas.
Objetivo: Ver estadísticas sobre el uso de las pistas de pádel.
Actor(es): Administrador.
Precondición: El administrador debe haberse autenticado.
Pasos: 1. El administrador accede las *estadísticas*. 2. Ve las estadísticas de las reservas.
Extensiones:

Caso de Uso: Registrarse.
Objetivo: Usuario sin registrar se registre en la aplicación para poder utilizarla.
Actor(es): Usuario visitante.
Precondición:
Pasos: 1. Visita la aplicación sin ser usuario registrado. 2. Accede a la interfaz de registro. 3. Introduce los datos personales del usuario nuevo. 4. Se registra.
Extensiones: 3.3. Introduce un usuario que no exista ya en la aplicación. 3.3.1. Si introduce un usuario ya existente el sistema le avisa de que no puede introducir ese nombre de usuario. 3.4. Introducir una contraseña válida. 3.4.1. Si introduce una contraseña inválida el sistema le avisa de que no puede introducir esa contraseña. 3.5. Todos los demás campos son obligatorios. 3.5.1. Si no introduce valores en los campos requeridos o introduce valores no válidos se le indicará que no puede introducir esos valores.

Caso de Uso: Reservar pista.
Objetivo: Reservar una pista en una fecha y hora deseadas.
Actor(es): Usuario registrado.
Precondición: El usuario registrado debe estar autenticado.
Pasos: 1. El usuario registrado accede a la sección de *ver disponibilidad de las pistas*. 2. Elige la pista deseada. 3. Elige la fecha y hora deseada. 4. Reserva la pista.
Extensiones: 3.3. La pista ya está reservada ese día a esa hora. 3.3.1. El sistema le avisa que esa pista en ese momento ya está reservada y que no puede reservarla. 4.1. Si el usuario introduce un número de jugadores inferior a 4 el sistema enviará una notificación a aquellos usuarios del mismo nivel que el usuario registrado y que tengan activadas las notificaciones. 4.1.1. Los usuarios que reciben la notificación pueden unirse al partido o rechazarlo.

Caso de Uso: Modificar datos personales.
Objetivo: Modificar los datos personales del usuario.
Actor(es): Usuario registrado.
Precondición: El usuario registrado debe estar autenticado.
Pasos: 1. El usuario registrado accede a ver sus *datos personales*. 2. Modifica los datos personales que desee. 3. Confirma los datos modificados.
Extensiones: 3.1. Introduce valores inválidos. 3.1.1. El sistema le avisa de que está introduciendo datos inválidos para que los modifique.

Caso de Uso: Ver historial de sus reservas.
Objetivo: Ver el historial de todas las reservas realizadas por el usuario registrado.
Actor(es): Usuario registrado.
Precondición: El usuario registrado debe estar autenticado.
Pasos: 1. El usuario registrado accede a la sección del *historial de sus reservas realizadas*. 2. Ve la lista de reservas realizadas por él.
Extensiones:

Caso de Uso: Apuntarse a partido.
Objetivo: Apuntarse a un partido que ya haya sido creado por otro usuario de su mismo nivel.
Actor(es): Usuario registrado.
Precondición: El usuario registrado debe estar autenticado y el partido ha de estar creado por otro usuario de su nivel.
Pasos: 1. Cuando el usuario registrado recibe la notificación pincha en ella para acceder a la información del partido. 2. Acepta la invitación al partido.
Extensiones:

Caso de Uso: Elegir si quiere notificaciones.
Objetivo: Elegir si quiere o no recibir notificaciones de la aplicación.
Actor(es): Usuario registrado.
Precondición: El usuario registrado debe estar autenticado.
Pasos: 1. El usuario registrado accede a la sección de *modificación de sus datos personales*. 2. Selecciona si quiere notificaciones o no. 3. Guarda los cambios realizados.
Extensiones:

7. Diseño

En esta sección describiremos los aspectos relacionados con la fase de diseño de nuestra aplicación. En particular, se mostrará el diseño de los prototipos de las interfaces de usuario y el diseño del esquema de la base de datos.

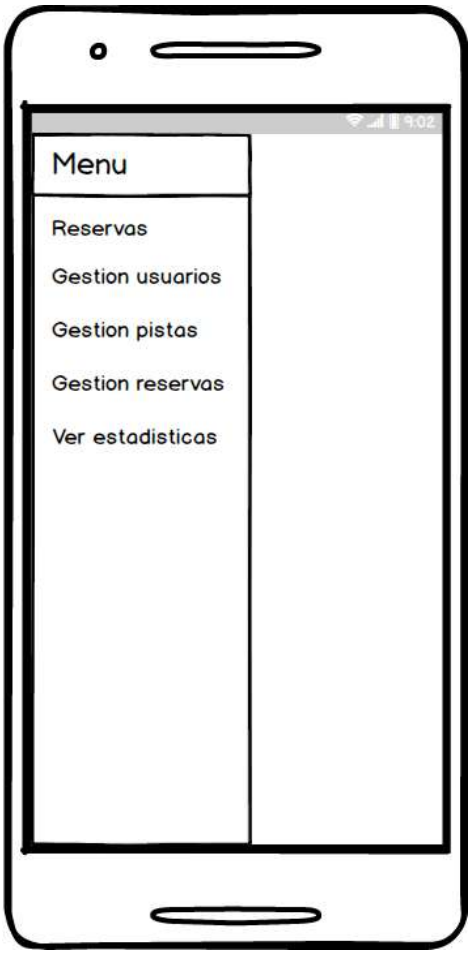
7.1. Prototipos de las interfaces

En este apartado se muestran los prototipos de las interfaces de usuario que va a tener la aplicación (tanto web como móvil). Se han dividido en dos partes, que a su vez están divididas en otras dos partes. Primero se diferencia entre interfaces para usuarios y para el administrador, y cada uno de ellos tendrá la versión móvil y la versión web.

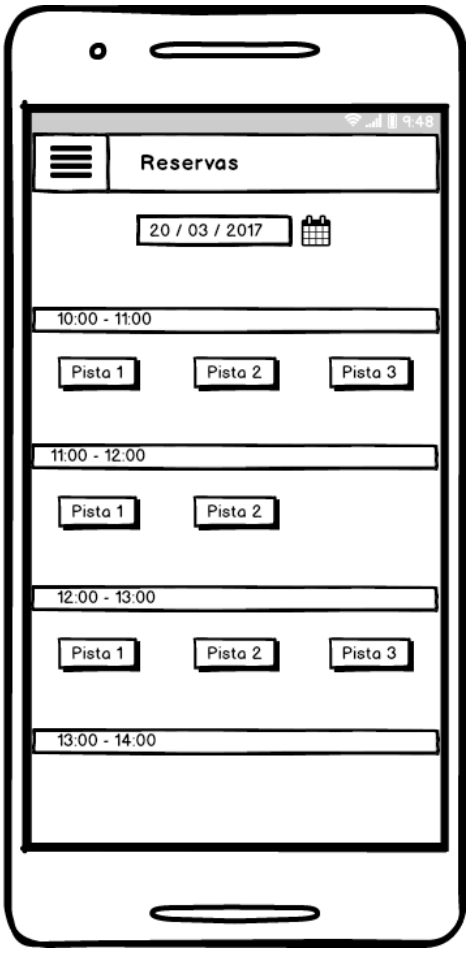
7.1.1. Prototipos de interfaces de la versión móvil y web para el administrador

Ambas versiones del *administrador* van a tener las mismas interfaces con la única diferencia de que unas van a ser en versión móvil y otras en versión web, pero ambas van a tener la misma estructura y los mismos elementos. El *administrador* va a tener una cuenta ya creada donde sólo se guardará el email y la contraseña con la que va a acceder a la aplicación. Por ello, no habrá ninguna interfaz de registro para el *administrador*. Una vez que el *administrador* haya iniciado una sesión en la aplicación, los prototipos de las interfaces que le aparecerán serán las que se presentan a continuación.

La aplicación dispondrá de un menú desplegable en la parte izquierda superior de la pantalla  , que se repetirá en todas las pantallas, y con el que podremos acceder a todas las demás funcionalidades de la aplicación (ver imagen de la izquierda en la siguiente pareja de figuras). Además de esto, la interfaz principal de la aplicación será la de realización de reservas. Desde esta pantalla podrá ver la lista de las pistas que no estén reservadas, considerando la fecha actual por defecto. Cuando el *administrador* pulse en una pista para reservarla, éste tendrá que introducir el nombre de usuario del usuario que desea reservar la pista. Por este motivo, el nombre de usuario de cada usuario registrado tiene que ser único (ver imagen de la derecha en la siguiente pareja de figuras).

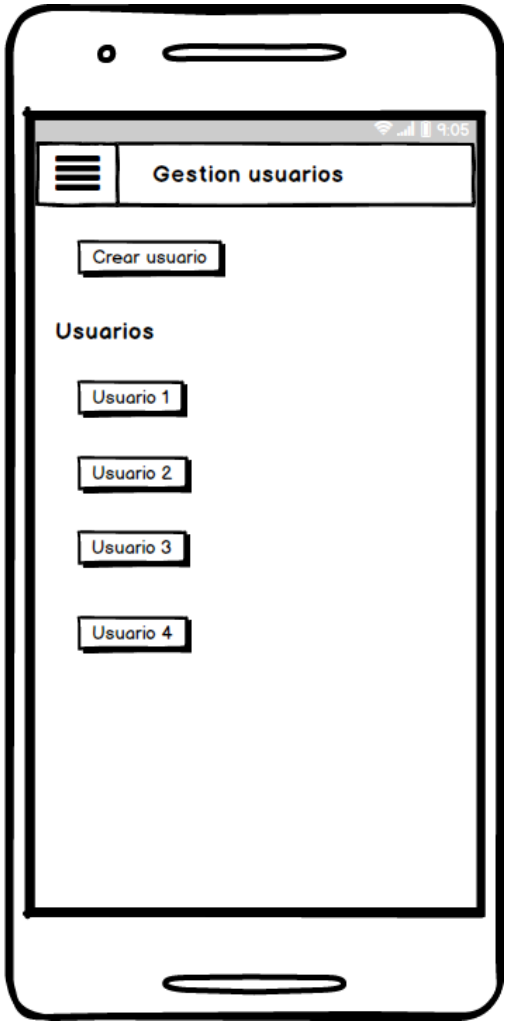


Menú desplegable.

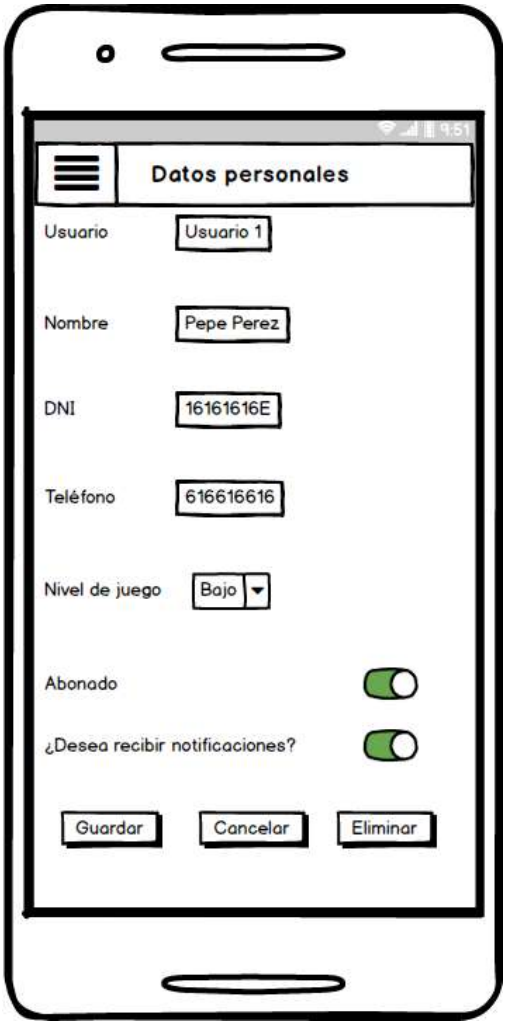


Pantalla de reservas (principal).

El *administrador* puede gestionar los usuarios, las reservas y las pistas accediendo desde el menú desplegable mostrado anteriormente. Cuando el *administrador* acceda a la "Gestión usuarios" verá la lista de usuarios registrados en la aplicación (ver la imagen de la izquierda en la siguiente pareja de figuras), El *administrador* podrá pulsar sobre un usuario para modificar sus datos personales o incluso eliminar el usuario (ver imagen de la derecha en la siguiente pareja de figuras).



Pantalla de gestión usuarios.



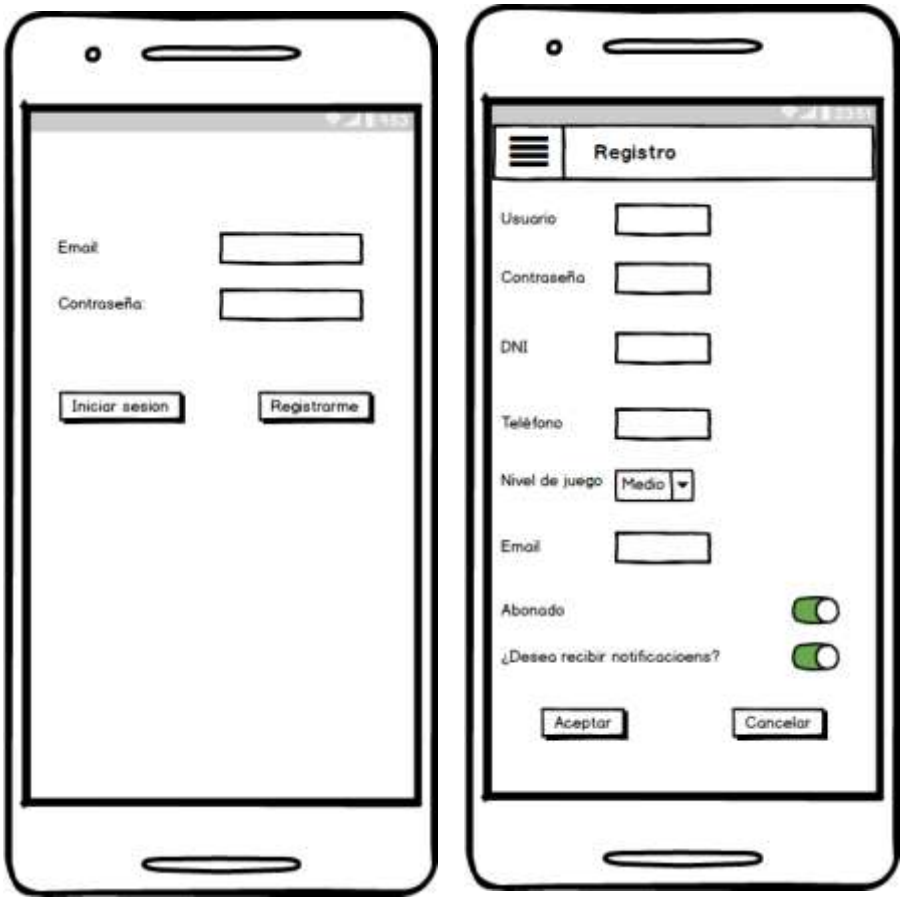
Pantalla para modificar los datos de un usuario.

Las interfaces de "Gestión pistas" y de "Gestión reservas" son similares a la de "Gestión usuarios" con la diferencia de que en la de "Gestión reservas" la lista será de reservas que hay realizadas (activas o no) y en la "Gestión pistas" habrá una lista con las pistas dadas de alta en la aplicación. Además, al seleccionar una pista o reserva, el *administrador* también podrá eliminar dicha pista o reserva.

Por último, en la interfaz para ver las estadísticas de la aplicación, el *administrador* verá cuales son las pistas más reservadas y cuáles son las horas en las cuales se hacen más reservas. A esta interfaz se accede desde el menú desplegable.

7.1.2. Interfaces de la versión móvil para usuarios registrados

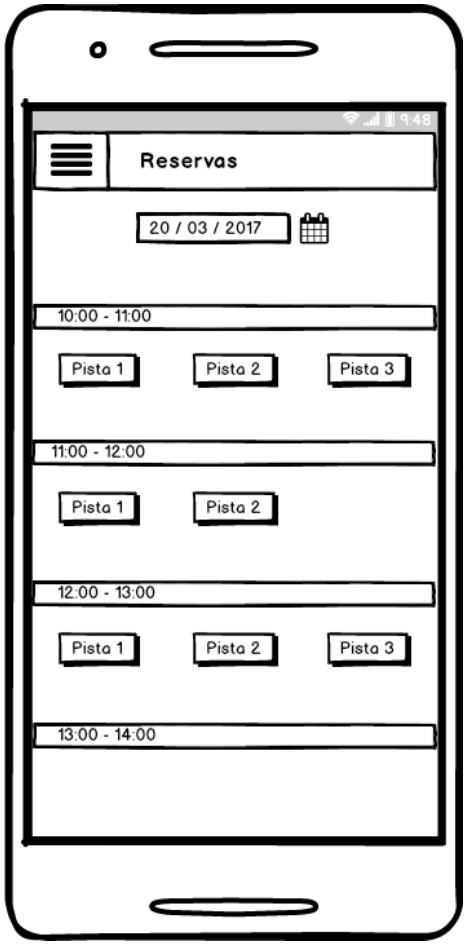
La primera pantalla que aparecerá al acceder a la aplicación, en el caso de que el *usuario* no haya iniciado sesión, será la que se muestra a continuación en la imagen de la izquierda. En caso de que el *usuario* no tenga una cuenta creada, podrá seleccionar el botón “Registrarme” a través del cual accederá a la pantalla de la derecha, que es la interfaz de “Registro” donde podrá introducir sus datos y así crear un nuevo usuario.



Pantalla de inicio de sesión. Pantalla de registro.

Cuando el *usuario registrado* se haya autenticado, al igual que en la aplicación para el *administrador,* verá directamente la interfaz que implementa la funcionalidad principal de la aplicación, es decir, la de realización de reservas (ver la imagen situada a la izquierda en la siguiente página).

Además, la aplicación también dispondrá de un menú desplegable al igual que el de la aplicación para el *administrador,* pero con diferentes botones a las funcionalidades disponibles para el *usuario registrado* (ver imagen de la derecha).



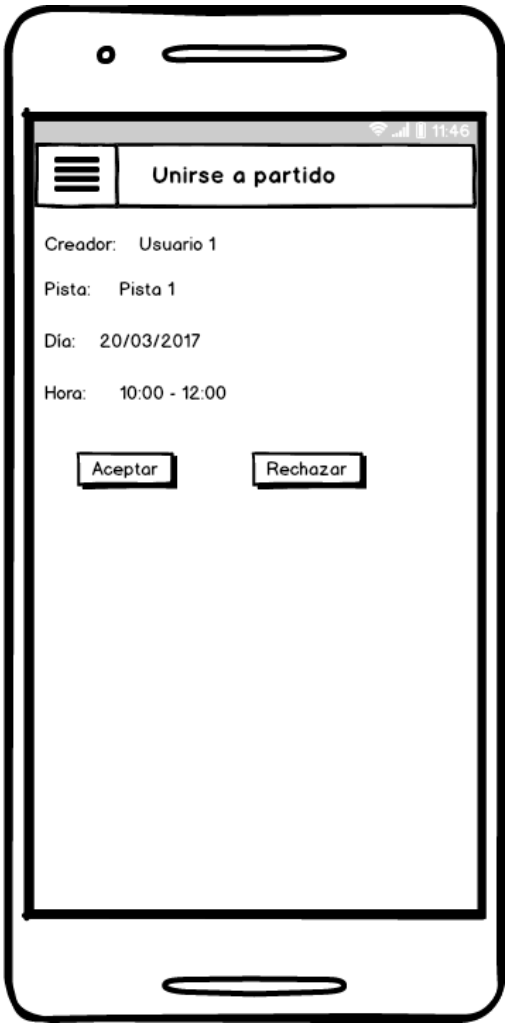
Pantalla de reservas (principal).

Menú principal de la aplicación.

Cuando el *usuario registrado* quiera reservar una pista a una hora y un día determinados, elegirá en la pantalla principal de la aplicación el día y la hora deseada. Posteriormente, pinchará sobre la pista que quiere reservar siempre y cuando esta pista aparezca en esa hora y día. Si no aparece, significará que ya está reservada. En ese momento, la aplicación llevará al *usuario registrado* a la interfaz de "Reservar" (ver la imagen de la izquierda de la pareja de imágenes que se muestra en la siguiente página). En esta pantalla aparecerán ya rellenos los campos con el nombre de usuario, el nombre de la pista, el día y la hora de inicio de la reserva. El *usuario registrado* no podrá modificar estos datos. Los únicos datos que podrá modificar son el número de jugadores abonados y no abonados. Si finalmente desea reservar la pista, le dará al botón de "Reservar".

Si el *usuario registrado* ha introducido un número de jugadores inferior a cuatro, el sistema enviará una notificación a todos los usuarios registrados que tengan activadas las notificaciones y cuenten con el mismo nivel de juego que el usuario que está haciendo la reserva, para que estos usuarios puedan unirse al partido hasta que éste esté completo. Cuando un *usuario registrado* visualice una notificación, accederá a la interfaz correspondiente para unirse al partido si así lo desea (ver la imagen de la derecha).

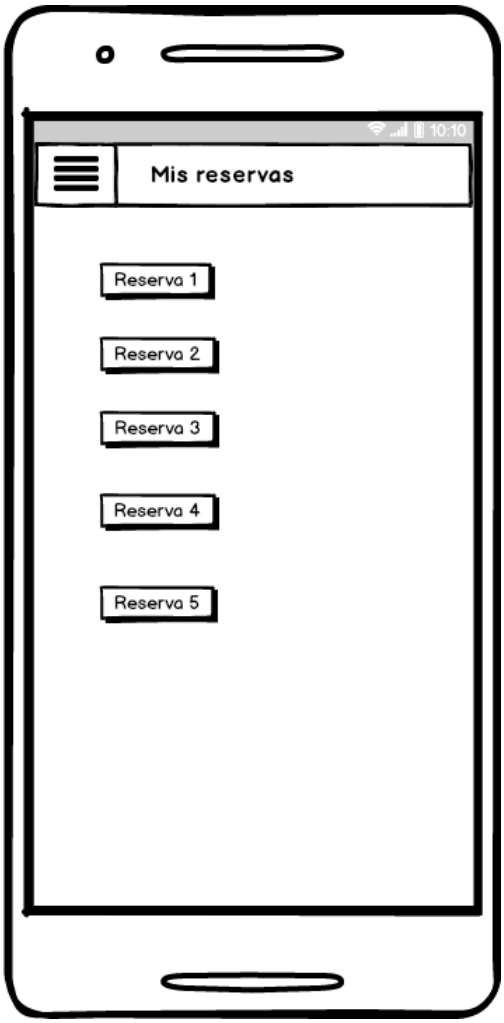
Pantalla para reservar una pista.



Pantalla para unirse a un partido.

Los *usuarios registrados* en la aplicación podrán modificar sus datos personales accediendo a la interfaz de “Datos personales” desde el menú desplegable que aparece en la parte izquierda superior de cada pantalla (ver imagen de la izquierda de la pareja de imágenes siguiente).

Los *usuarios registrados* también podrán ver un historial con todas las reservas que han realizado a través de la aplicación (ver imagen de la derecha de la pareja de imágenes siguiente).



Pantalla de modificación de datos personales. Pantalla para ver el historial de reservas.

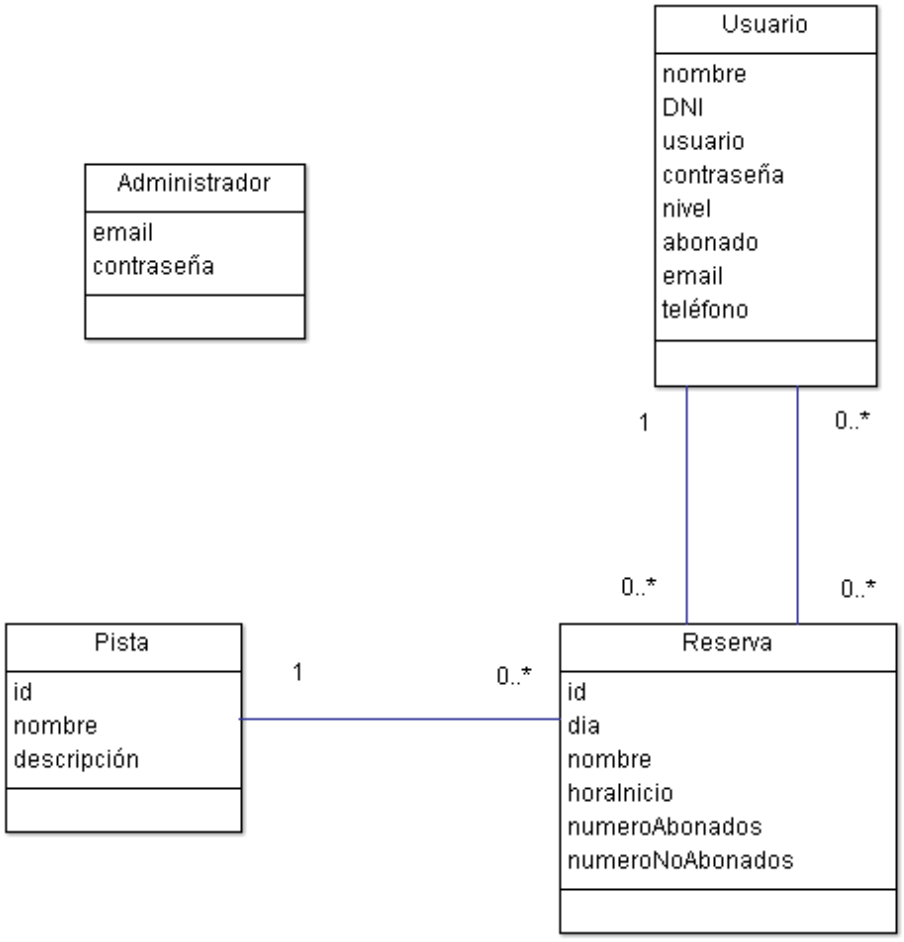
7.1.3. Prototipos de las interfaces de la versión web para usuarios registrados

Esta versión contará con las mismas pantallas que la versión móvil, con excepción de la pantalla de "Unirse a partido" que, como se ha comentado anteriormente, sólo va a aparecer en la versión móvil. Estos prototipos de interfaces se pueden ver en el anexo que hay al final del documento.

7.2. Diseño del esquema de la base de datos

Como se ha comentado anteriormente, *Firebase* almacena los datos en una base de datos *NoSQL,* mediante ficheros *JSON,* y por lo tanto no vamos a contar con tablas relacionadas. No obstante, se ha decidido diseñar un diagrama de clases que simule el esquema de lo que sería la BD relacional que almacenaría los datos de la aplicación, con objeto de ayudarnos a definir el esquema de los archivos *JSON*.

A continuación, se muestra el diseño de la BD:



.

En la clase “Administrador” sólo se va a almacenar un email (“email”) y una contraseña (“contraseña”), correspondiente al email y la contraseña con la que el administrador se va a autenticar en la aplicación. Esta clase no está relacionada con ninguna otra, puesto que solo tendremos un administrador que gestione todos los demás elementos de la aplicación.

En la clase “Usuario” se va a almacenar también el email (“email”) y contraseña (“contraseña”) del usuario, que son los valores necesarios para que los usuarios se autentiquen en la aplicación. También se almacena un nombre de usuario en el atributo “usuario”, que va a ser único y diferente para cada usuario. Además, de cada usuario se registrarán datos personales como el DNI (“DNI”), el nivel (“nivel”), su nombre (“nombre”), si es abonado o no (“abonado”) y su teléfono (“teléfono”).

La clase “Reserva” está relacionada con la clase “Usuario” de forma que un usuario puede tener ninguna o muchas reservas, y una reserva solo puede ser realizada por un usuario. Esta clase también está relacionada con la clase “Pista” de forma que se pueden realizar muchas reservas de una pista y una reserva solo puede ser de una pista.

En la clase “Pista” solo se almacena el identificador único de cada pista (“id”), su nombre (“nombre”) y una pequeña descripción de la pista (“descripción”).

8. Implementación

En esta sección se explican y detallan los aspectos y pasos más importantes realizados para implementar el proyecto.

La intención inicial en este proyecto ha sido utilizar la arquitectura recomendada por *AngularJS*, que es un patrón MVC (modelo, vista, controlador). Como es sabido, este patrón separa la aplicación en tres capas: la *capa de persistencia* (modelo), la *capa de lógica de negocio* (controlador) y la *capa de presentación* (vista).

La *capa de persistencia* es donde residen los datos y es la encargada de acceder a los mismos. La *capa de lógica de negocio* es donde residen los programas que se ejecutan, se reciben las peticiones del usuario y se envían las respuestas tras el proceso. Esta capa se comunica con la *capa de presentación*, para recibir las solicitudes y presentar los resultados, y con la *capa de persistencia*, para solicitar al gestor de base de datos almacenar o recuperar datos de él. La *capa de presentación* es con la que interactúa el usuario y envía las peticiones que desea el usuario a la *capa de lógica de negocio* y espera el resultado para mostrárselo al usuario [18].

No obstante, nos hemos visto obligados a mezclar en una sola capa la *capa de persistencia* y la de *lógica de negocio* debido a diversos problemas con la tecnología que se explican más adelante. Por ello, nos ha resultado más sencillo unir estas dos capas en una. Solamente hemos podido seguir la división en tres capas en el caso de las funciones y los servicios de autenticación usados de *Firebase*.

Notaremos que la preparación del entorno para el desarrollo de la aplicación se encuentra en el Anexo 12.2. “Preparación del entorno”.

8.1. Preparación de la base de datos

La base de datos va a estar almacenada en línea en *Firebase* [16]. Para crear la base de datos, lo que hay que hacer en primer lugar es crear una cuenta de correo electrónico de *Google* (*Gmail*) y acceder a *Firebase* desde la página web [16]. Desde ahí accederemos a nuestra cuenta en la que se almacenarán los proyectos con los que estamos trabajando, como podemos ver en la siguiente imagen.

Como se muestra en la anterior imagen, en nuestra cuenta hay creados dos proyectos diferentes llamados 'Padel' y 'prueba'. Al acceder a cada uno de ellos se puede ver los diferentes servicios que ofrece *Firebase* nombrados en la Sección 4. "Tecnologías a utilizar". Debemos configurar dichos servicios.

- *Firebase Auth*: dentro de este servicio podremos elegir de qué forma queremos que los usuarios se autentiquen dentro de nuestra aplicación. En nuestro caso, hemos habilitado la opción para que los usuarios accedan por medio de un correo electrónico y una contraseña. Para registrar un nuevo usuario y autenticarlo se utiliza un *proveedor* que tendremos que crear para usar la funcionalidad que nos da *Firebase.* Este servicio también crea un identificador único para cada usuario ("uid"). La creación de dicho proveedor es la siguiente:

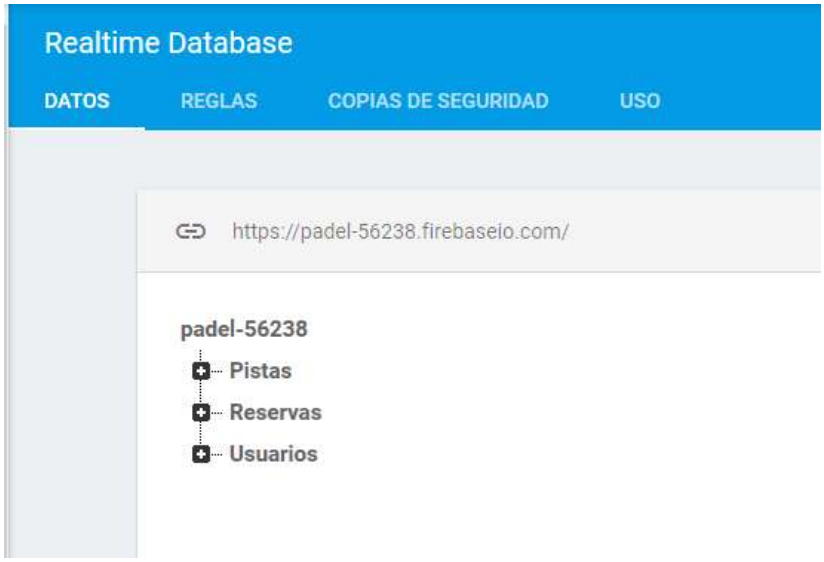
```
export class AuthProvider{
  constructor(public af:AngularFire) {}

  //Obtiene el id creado para cada usuario al registrarlo en la aplicacion
  getCurrentUid(){
    return this.af.auth.getAuth().auth.uid;
  }

  //Sirve para autenticarse dentro de la aplicación mediante usuario y contraseña
  signin(credentails) {
    return this.af.auth.login(credentails);
  }

  //Crea un nuevo usuario dentro de la aplicación con usuario y contraseña
  createAccount(credentails) {
    return this.af.auth.createUser(credentails);
  };

  //Cierra la sesión del usuario
  logout() {
     this.af.auth.logout();
  }

}
```

- *Realtime database*: a través de este servicio es como realmente se almacenan los datos de la aplicación (en nuestro caso los *usuarios*, las *pistas* y las *reservas*). La estructura de almacenamiento se puede crear de dos maneras:
    - *Manualmente,* introduciendo las estructuras y subestructuras a mano desde el servicio de la página de *Firebase*.
    - *Dinámicamente*, añadiendo los datos a medida que se va ejecutando la aplicación e insertando los datos. El problema que tiene esta opción es que siempre se tiene que seguir la misma estructura de datos con exactamente los mismos nombres, diferenciando entre mayúsculas y minúsculas, porque si no puede generar estructuras diferentes que pueden dar muchos problemas a la hora de testear el código.

Por todo esto es por lo que hemos realizado anteriormente un diseño del esquema de la base de datos, para que, cuando tengamos que introducir nuevos elementos en la base de datos, siempre insertemos los datos con la misma estructura y con los mismos nombres.

Para sincronizar este servicio con el servicio de autenticación comentado anteriormente, hemos tenido que almacenar como parte de cada usuario un nuevo atributo llamado `uid`, que es la copia del que genera y almacena el servicio de autenticación cuando se crea un usuario nuevo en la aplicación.

Para almacenar los datos en la base de datos, hemos elegido una mezcla entre las dos formas de introducir los datos. En primer lugar se han creado manualmente desde la aplicación las tres grandes estructuras de datos que son `Usuarios`, `Pistas` y `Reservas`.



En segundo lugar, se ha utilizado la estrategia de creación dinámica para insertar los datos en esas estructuras, tal y como se explica en la siguiente sección.

8.2. Inserción de datos en la base de datos

Una vez creadas las tres estructuras del apartado anterior, se van introduciendo los datos dentro de las mismas a medida que la aplicación va ejecutándose. Para ello, en primer lugar tenemos que importar el módulo de *AngularFire* en las clases de la *lógica de negocio* desde las que queremos enviar o recibir datos de la base de datos. Dicha importación se hace de la siguiente forma:

```
import { AngularFireModule, AuthMethods, FirebaseListObservable, AngularFire } from 'angularfire2';
```

Como se puede observar en el código anterior, añadimos los diferentes módulos de *AngularFire* necesarios para la conexión con la base de datos.

Ahora que tenemos ya importados los módulos necesarios, veremos con un caso concreto como insertar nuevos datos. Por ejemplo, si queremos crear una nueva pista, lo primero que tenemos que hacer es crear una referencia a la base de datos, en concreto, a la estructura llamada '*Pistas'* que es donde vamos a introducir todas las pistas. Para ello:

- En primer lugar creamos una variable del tipo `FirebaseListObservable<any>` que es la que va a contener la referencia a la base de datos:

```
pistas: FirebaseListObservable<any>;
```

- En segundo lugar usamos un método que nos proporciona *AngularFire* para conseguir la referencia a las pistas:

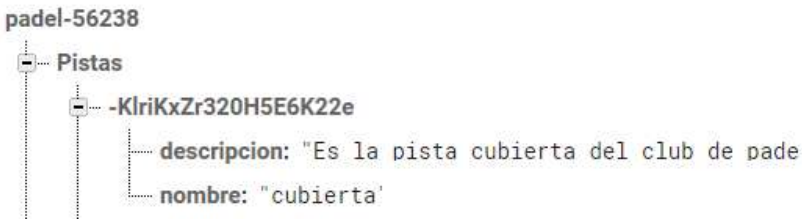
```
constructor(public navCtrl: NavController,
            public navParams: NavParams,
            af: AngularFire,
            public auth: AuthProvider,
            public toastCtrl: ToastController
            )
{
  this.pistas = af.database.list('/Pistas');
```

- Por último, para añadir la pista que deseamos con sus datos, utilizamos la función `push` de la referencia de la siguiente forma:

```
this.pistas.push({
    nombre: datos.nombre,
    descripcion: datos.descripcion
});
```

Lo que conseguimos con esto es crear una nueva pista con un nombre definido por el elemento `datos.nombre` y una descripción definida por `datos.descripcion`.

Ésto nos genera en la base de datos de *Firebase* lo siguiente:



Como podemos observar, dentro de la estructura creada anteriormente llamada `Pistas`, se ha creado una nueva pista, con una clave única asociada a la nueva pista. A esta clave, se le podrá hacer referencia posteriormente a través de la variable `$key`.

8.3. Recuperación de los datos de la base de datos

En *Firebase* h*ay* diferentes formas de recuperar los datos dependiendo de si la información que se quiera recuperar tiene que estar o no filtrada por algún criterio.

Si lo que deseamos es recuperar todos los elementos de una estructura, como por ejemplo todas las pistas que hay almacenadas en la base de datos, lo que debemos hacer es utilizar *AngularFire* para devolver en una variable `FirebaseListObservable<any>` todos los elementos que hay dentro de la referencia de la base de datos llamada `Pistas`.

En el ejemplo se haría de la siguiente forma:

```
pistas: FirebaseListObservable<any>;

constructor(public navCtrl: NavController,
            public alertCtrl: AlertController, public af: AngularFire,
            public actionSheetCtrl: ActionSheetController,
            public auth:AuthProvider,
            ) {
                this.pistas = this.af.database.list('/Pistas');
}
```

Como se puede apreciar, en la variable `pistas` tendremos todos los elementos que haya dentro de la referencia `/Pistas`.

Si por el contrario tan solo queremos recoger los elementos que cumplan una condición en concreto dentro de la base de datos, como por ejemplo coger las reservas que hayan sido reservadas por el usuario que está actualmente utilizando la aplicación, tenemos que usar la función `query` de la siguiente forma:

```
this.reservas = af.database.list('/Reservas', {
  query: {
    orderByChild: 'usuario',
    equalTo: this.us
  }
})
```

Como vemos en el código anterior, recuperamos una referencia a todas las reservas, pero ahora utilizamos la función `query`. Las reservas van a tener un atributo llamado `usuario` donde se almacena el nombre del usuario que ha realizado la reserva, el cual es único. Posteriormente lo que hacemos es ordenar las reservas por el atributo `usuario` y luego coger los que sean igual al nombre de usuario que está actualmente utilizando la aplicación.

8.4. Uso de los datos de la base de datos

En este apartado se muestran algunos aspectos del uso de los datos de la base de datos desde la *capa de presentación* y desde la capa de *lógica de negocio*.

- Representar los datos en una vista (*capa de presentación*). Una vez que tenemos, por ejemplo, la lista de las reservas que ha realizado el usuario, podemos mostrar en la vista una lista con todas las reservas del usuario y sus datos más significativos.

  Para ello, utilizamos la característica de *AngularJS* que nos permite el paso bidireccional de datos entre la vista y el controlador, utilizando una variable pública creada en el controlador. Para mostrarlos en la vista se haría de la siguiente forma:

```
<ion-list *ngFor="let res of reservas | async">
  <ion-item>
      <ion-label >Nombre</ion-label>
      <ion-input type="text" id="nombre"  value="{{res.nombre}}" readonly="readonly"></ion-input>
  </ion-item>
  <ion-item>
      <ion-label >Dia</ion-label>
      <ion-input type="text" id="dia"  value="{{res.dia}}" readonly="readonly"></ion-input>
  </ion-item>
  <ion-item>
      <ion-label >Hora inicio</ion-label>
      <ion-input type="text" id="inicio"  value="{{res.horaInicio}}" readonly="readonly"></ion-inp
  </ion-item>
  <ion-item>
      <ion-label >Usuario</ion-label>
      <ion-input type="text" id="usuario"  value="{{res.usuario}}" readonly="readonly"></ion-input
  </ion-item>
  <ion-item>
      <ion-label >Pista</ion-label>
      <ion-input type="text" id="pista"  value="{{res.nombrePista}}" readonly="readonly"></ion-inp
  </ion-item>
</ion-list>
```

  Como vemos con la función `*NgFor` de *AngularJS* se recorre la variable `reservas`, que tiene que estar creada como variable pública en la clase del controlador. Cada vez que se trata una reserva, como vemos, recoge los datos deseados poniendo los valores entre doble llaves ({{}}).

- Otro uso que le hemos dado a los datos recuperados de la base de datos es recorrerlos en los controladores para manejarlos de uno en uno (*capa de lógica de negocio*).

Para esto, tenemos que utilizar el método `forEach` de las variables `FirebaselistObservable` para recorrer todos los elementos de la variable del siguiente modo:

```
this.pistas.forEach(pista => {
  this.mostrarReservas();
});
```

En este ejemplo lo que hacemos es, por cada pista que hay almacenada en la variable `pistas`, se llama al método `mostrarReservas()`.

8.5. Modificación de los datos

En esta sección vamos a explicar cómo modificar los datos almacenados en la base de datos. Lo ilustraremos modificando los datos personales de un usuario. Lo primero que tenemos que hacer es obtener la referencia al dato a modificar.

Para modificar un dato concreto de un usuario almacenado en la base de datos, tenemos que recorrer todos los usuarios de la base de datos y obtener el usuario que está actualmente conectado, buscándolo por el atributo `uid` del usuario:

```
this.usuario = this.af.database.list('/Usuarios', {
    query: {
        orderByChild: 'uid',
        equalTo: this.user.uid
    }
});
```

Cuando ya tenemos el objeto `usuario` del que queremos modificar sus datos, debemos obtener la clave asociada cuyo valor se obtiene accediendo al valor de `$key` del usuario correspondiente. Con dicha clave podemos obtener la referencia en la base de datos *Firebase* al usuario específico que queremos modificar:

```
this.referencia = firebase.database().ref('Usuarios/' + u.$key);
```

Teniendo la referencia al elemento en concreto de la base de datos (`referencia`), ahora podemos modificar los atributos que deseemos utilizando el método `update` del siguiente modo:

```
this.referencia.update({
      nombre: d.nombre,
      dni: d.dni,
      telefono: d.telefono,
      abonado: ab.lastChild.attributes.item(6).nodeValue,
      nivelJuego: d.nivel,
      usuario: d.usuario
})
```

Con el anterior código lo que hacemos es modificar el valor de los atributos que aparecen en la parte izquierda de los dos puntos y les asignamos el valor que hay a la derecha. En caso de que no haya un atributo con el mismo nombre al que aparece dentro del `update,` se crea un atributo con dicho nombre dentro del elemento referenciado. Por ello hay que tener mucho cuidado y poner el nombre del atributo exactamente igual al nombre del atributo que hay en la estructura correspondiente de la base de datos.

8.6. Principales problemas encontrados

Haremos notar que durante la implementación se han tenido muchos problemas, casi todos derivados de ser primerizo en la tecnología y del desconocimiento de los lenguajes utilizados en el proyecto. Nunca antes había utilizado ninguno de los lenguajes usados en la implementación del proyecto a excepción de *HTML* y *CSS* (que justo coincide con los lenguajes que menos he utilizado debido a su pequeño papel dentro de la aplicación).

HTML no se ha utilizado mucho debido a que *Ionic* proporciona elementos dentro de las vistas para crearlo. Igualmente *CSS* tampoco se ha utilizado mucho debido a que la aplicación desarrollada dispone de una interfaz muy sencilla e intuitiva en la que no se han introducido muchos elementos de estilo y diseño.

El principal problema con el que me he encontrado durante toda la implementación ha sido el hecho de que la ejecución del código del programa se realiza de forma asíncrona. Esto quiere decir que el código del programa no se ejecuta de forma secuencial y ordenada, sino que no se sabe en qué orden se van a ejecutar las líneas de código. Esto es porque las funciones asíncronas se ejecutan en segundo plano mientras que el programa se sigue ejecutando sin esperar a que la función asíncrona devuelva el resultado.

Esta forma de ejecución sirve para mejorar el rendimiento de la aplicación, puesto que el programa sigue ejecutando líneas de código sin esperar a los resultados de las funciones, las cuales se siguen ejecutando en segundo plano, evitando ralentizar la ejecución del programa.

Este aspecto me ha creado un verdadero quebradero de cabeza durante toda la implementación de la aplicación, puesto que, al no controlar la ejecución del código y al ejecutarse con un orden aleatorio, muchas de las partes del código daban error. Esto ha sido debido a que muchas veces ejecutaba una función cuyo valor devuelto iba a necesitar más adelante, pero al intentar tratar ese valor, el código daba error puesto que el código trataba el valor antes de que la función terminase su ejecución y por lo tanto el valor era indefinido.

Uno de los mayores problemas con este tema lo he tenido al recoger datos de la base de datos de *Firebase* desde los controladores, e intentar tratar esos datos desde los mismos. Al hacerlo, la ejecución no esperaba a que la función de recuperación de los datos de la base de datos terminase de devolverme los datos necesarios para poder usarlos, por ejemplo en la *capa de presentación*.

Este problema lo tuve, por ejemplo, al intentar mostrar las pistas en las diferentes horas del día, en un día determinado, siempre y cuando no haya una reserva de esa pista realizada a esa hora y ese día. Esto me llevó mucho tiempo solventarlo. Intenté ir depurando el código mostrando los resultados del código en el log de la ejecución del programa, pero lógicamente, tampoco controlaba cuándo se ejecutaba el código de envío de mensajes al log. Algunas veces según el log el código estaba bien escrito, pero el código no hacía lo deseado.

Al final conseguí solucionar este problema utilizando la función `subscribe` en cada llamada a la base de datos. Este método hace que la ejecución del código espere a la respuesta de esta función y no continúe con la ejecución del resto del código. Ésto solventaba el caso de recoger datos de la base de datos una sola vez, pero en el controlador de la vista principal de la aplicación necesitaba hacer más de una llamada a la base de datos para recogerlos, por lo tanto, no era suficiente con esta función. Por ello, lo que tuve que hacer es añadir una variable de tipo `booleano` por cada llamada a la base de datos e inicializarla a `false`. Una vez inicializadas, dentro de cada método `subscribe`, a estas variables les daba el valor `true` para así, al comprobar el valor de cada variable, saber si se habían recogido todos los datos de la base de datos. De esta forma, solo se ejecuta el método deseado cuando todas las variables tengan el valor de `true`, que significa que todas las llamadas a la base de datos han obtenido ya los datos. El código resultante queda de la siguiente forma:

```
this.usuarios.subscribe(items => {
  usuarioscargados = true;
  if(usuarioscargados && reservascargadas && pistascargadas && rescargadas){
    console.log('todo cargado');
    this.mostrarReservas();
  }
});
this.reservas.subscribe(items => {
  reservascargadas = true;
  if(usuarioscargados && reservascargadas && pistascargadas && rescargadas){
    console.log('todo cargado');
    this.mostrarReservas();
  }
});
this.pistas.subscribe(items => {
  pistascargadas = true;
  if(usuarioscargados && reservascargadas && pistascargadas && rescargadas){
    console.log('todo cargado');
    this.mostrarReservas();
  }
});
this.res.subscribe(items => {
  rescargadas = true;
  if(usuarioscargados && reservascargadas && pistascargadas && rescargadas){
    console.log('todo cargado');
    this.mostrarReservas();
  }
})
```

Como vemos en este ejemplo, se han tenido que recuperar usuarios, pistas y dos veces datos sobre reservas. Dentro de cada función `subscribe` ponemos la variable correspondiente a `true` y además comprobamos si todas las variables están puestas a `true` para que en el momento en el que estén todas con dicho valor, se ejecute la función `mostrarReservas()`. La llamada a la función `mostrarReservas()` la he tenido que poner dentro de todos los `subscribe` porque, si la ponía fuera de ellos, tenía el problema inicial de que se ejecutase antes de que todos los datos estuvieran cargados.

Otro de los mayores problemas que he tenido en este aspecto ha sido al querer mostrar en una vista los elementos recogidos en un controlador. Como ya hemos explicado anteriormente, esto se hace gracias a *AngularJS* que nos permite el paso bidireccional de información de datos entre la vista y el controlador. *AngularJS* nos proporciona funciones como por ejemplo `*NgFor` para recorrer listas dentro de las vistas.

No obstante al utilizar está función para mostrar los datos en la vista, al principio no se mostraba ningún dato, hasta que descubrí que el fallo estaba otra vez en la ejecución asíncrona y por lo tanto la vista intentaba mostrar datos sin estar todavía cargados en la variable que intentaba recorrer. Esto lo conseguí solucionar poniendo en la función `*NgFor` al final el valor `async` para que la vista esperase a que los datos estuvieran cargados por completo en la variable que la vista iba a recorrer. La instrucción concreta utilizada fue:

```
<ion-list *ngFor="let us of usuario | async">
```

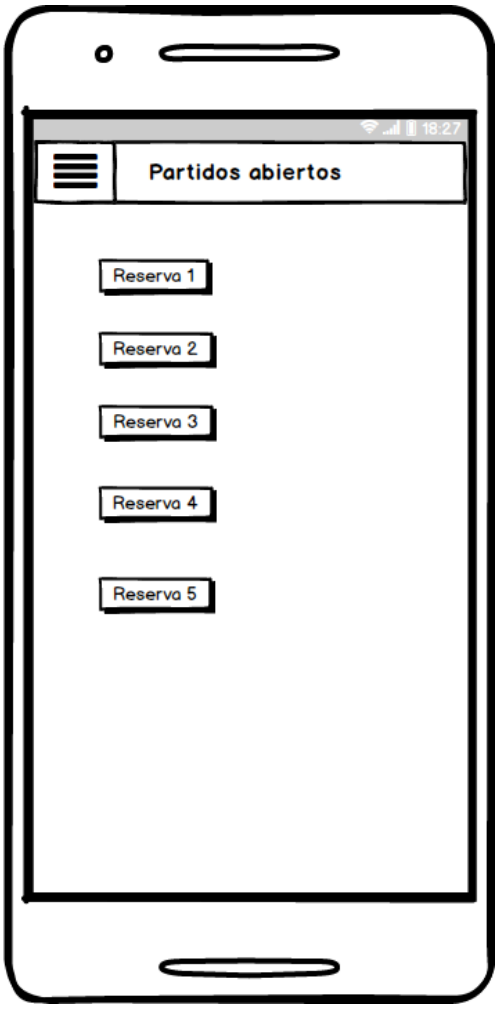
Todo lo anterior se presenta como una breve prueba de los problemas con los que me he tenido que enfrentar a lo largo del desarrollo del proyecto y sirven para justificar de alguna forma, como veremos más adelante, la desviación de horas que se ha sufrido. En el Anexo 12.3. “Otros problemas encontrados” se muestran otros problemas encontrados.

8.7. Requisitos no implementados

Como se ha comentado anteriormente, no he conseguido implementar la funcionalidad para el envío de notificaciones entre usuarios. Esto ha sido debido a que ha sido la última funcionalidad que he intentado desarrollar en la aplicación y, después de intentarlo durante muchas horas, no he visto resultados satisfactorios.

En particular en este caso he intentado especialmente buscar opciones alternativas. He buscado por internet en foros sobre el tema, he buscado en los propios foros de *Firebase,* e incluso, he tratado por correo con el propio soporte técnico de *Firebase,* que no me han dado ninguna solución.

Por ello, tras hablar con el cliente, se ha acordado no implementar esa funcionalidad porque iba a superar por mucho las horas máximas a dedicar al proyecto. Finalmente, utilizando las horas que me quedaban en la *fase de implementación,* se decidió modificar los requisitos iniciales para implementar una nueva funcionalidad. Esta funcionalidad consiste en que los *usuarios registrados* en sus versiones web y móvil puedan visualizar un listado de todas aquellas reservas de menos de cuatro jugadores que se han realizado en la aplicación (ver imagen siguiente). De esta forma, cuando un usuario se quiera apuntar a un partido pinchará sobre una reserva y el sistema le llevará a la pantalla ya creada de "Unirse a partido" (ver página 20).



9. Planificación real

En esta sección se muestra una tabla simplificada con (1) las horas que se habían estimado en la planificación para cada fase del proyecto, (2) las horas reales invertidas durante la realización del proyecto y por último (3) la desviación de las horas planificadas a las reales.

Fase	Horas planificadas	Horas reales	Desviación
1. Planificación	**6**	**6**	**0%**
2. Estudio de la tecnología	**20**	**30**	**50%**
3. Análisis de requisitos	**29**	**25**	**-14%**
4. Diseño	**45**	**30**	**-25%**
5. Implementación	**145**	**180**	**24%**
6. Pruebas	**10**	**10**	**0%**
7. Redactar memoria	**30**	**30**	**0%**
8. Reuniones con la tutora de la Universidad	**15**	**15**	**0%**
Total:	**300**	**326**	**8%**

En la *fase del estudio de la tecnología,* hemos superado las horas planificadas debido a que hemos trabajado con tecnologías nuevas con las que no habíamos trabajado nunca y nos ha resultado más difícil de aprender de lo que esperábamos a la hora de hacer la planificación.

En la *fase de análisis de requisitos* hemos utilizado menos horas de las planificadas. Esto ha sido debido a que hemos avanzado en esta fase más rápido de lo que esperábamos, ya que nos ha resultado más fácil analizar los requisitos de lo que esperábamos. Por ello, tendríamos que haber puesto menos horas a la hora de planificar.

En la *fase de diseño* también hemos utilizado menos horas de las planificadas porque la estructura de la base de datos era muy sencilla.

En la *fase de implementación*, la creación de la base de datos nos ha llevado menos tiempo del que estaba planificado. No obstante, el desarrollo de la funcionalidad del proyecto nos ha llevado mucho más tiempo del esperado debido al desconocimiento inicial de la tecnología y a todos los problemas derivados del uso de la misma (como por ejemplo el hecho del uso de funciones asíncronas).

En todas las demás fases se han cumplido las horas estimadas en la planificación inicial.

10. Conclusiones

Al concluir el proyecto, en cuanto a la funcionalidad que se deseaba inicialmente, decir que no se ha logrado lo esperado. No obstante acordándolo con el cliente, se ha conseguido contrarrestar ese problema implementando otra funcionalidad. Finalmente, se ha desarrollado un producto que, aunque no sea totalmente funcional, ha resultado ser una aplicación con la que el cliente está satisfecho.

Por lo tanto, en líneas generales, creo que ha sido un trabajo satisfactorio en el que he aprendido nuevos lenguajes de programación y un nuevo framework para la programación multiplataforma que creo que es el futuro del desarrollo de las aplicaciones.

Para el próximo proyecto tendré que analizar mejor la tecnología antes de iniciarlo e intentar cuadrar mejor las horas dentro de la planificación para que no haya (en general) tanto desvío de horas como ha habido en este proyecto.

Al haber desarrollado el proyecto en una empresa, mis compañeros y mi tutor de la empresa me han ayudado en todo lo posible durante todo el proyecto obteniendo un mejor resultado final del que hubiese conseguido si el proyecto lo hubiese realizado por mi cuenta. Me gustaría agradecerles su apoyo y ayuda.

11. Bibliografía

[1] *Ionic 2*. Página web: https://ionicframework.com/ [online]. Accedido en febrero de 2017.

[2] *Ionic 2*. Documentación. Disponible en https://ionicframework.com/docs/ [online]. Accedido en febrero de 2017.

[3] *AngularJS*. Página web: https://angular.io/ [online]. Accedido en febrero de 2017.

[4] *AngularJS*. Documentación. Disponible en https://angular.io/docs/ts/latest/ [online]. Accedido en febrero de 2017.

[5] *Typescript*. Página web: https://www.typescriptlang.org/ [online]. Accedido en febrero de 2017.

[6] *Typescript*. Documentación. Disponible en https://www.typescriptlang.org/docs/home.html [online]. Accedido en febrero de 2017.

[7] *NodeJS*. Página web: https://nodejs.org/es/ [online]. Accedido en abril de 2017.

[8] *Android Studio 2.3.3*. Página de descarga: https://developer.android.com/studio/index.html?hl=es-419 [online]. Accedido en mayo de 2017.

[9] *HTML 5*. Documentación. Disponible en https://www.w3schools.com/html/html5_intro.asp [online]. Accedido en febrero de 2017.

[10] *CSS*. Documentación. Disponible en: https://www.w3schools.com/css/ [online]. Accedido en febrero de 2017.

[11] *Visual Studio Code*. Página web: https://code.visualstudio.com/ [online]. Accedido en abril de 2017.

[12] *Github*. Página web: https://github.com/ [online]. Accedido en marzo de 2017.

[13] *Github*. Repositorio en el que está almacenado el código: https://github.com/seperev/aplicacion-final [online]. Accedido en marzo de 2017.

[14] *Balsamiq mockups*. Página web: https://balsamiq.com/products/mockups/ [online]. Accedido en marzo de 2017.

[15] *ArgoUML 0.34*. Página de descarga: https://argouml.uptodown.com/windows [online]. Accedido en marzo de 2017.

[16] *Firebase*. Página web: https://firebase.google.com/?hl=es-419 [online]. Accedido en febrero de 2017.

[17] *Firebase*. Documentación. Disponible en: https://firebase.google.com/docs/web/setup?hl=es-419 [online]. Accedido en febrero de 2017.

[18] Programación por capas. *Wikipedia*. Página web https://es.wikipedia.org/wiki/Programaci%C3%B3n_por_capas [online]. Accedido en febrero de 2017.

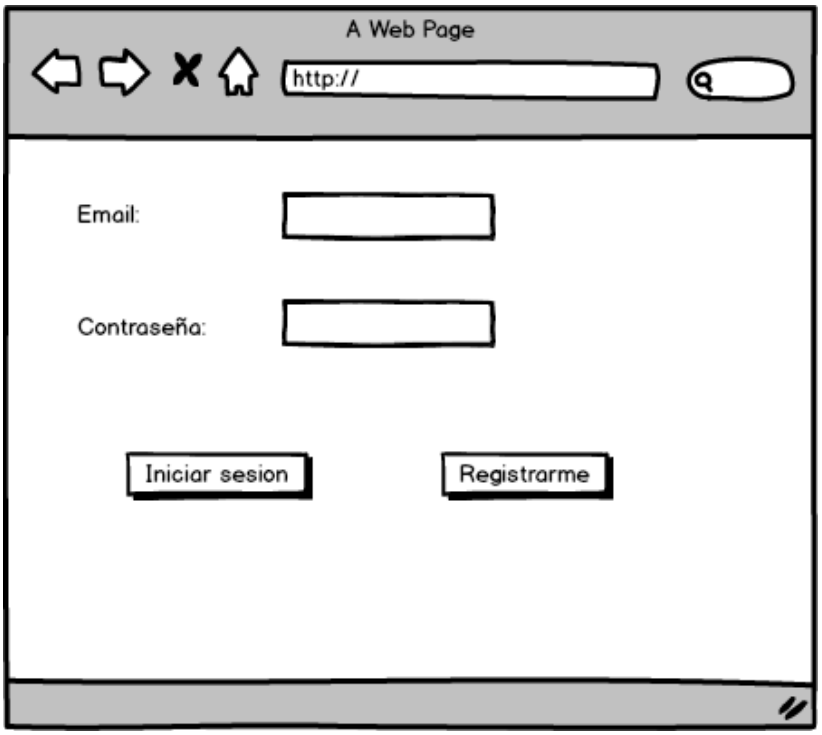
12. Anexos

En esta sección aparecen los anexos a la memoria de nuestro proyecto.

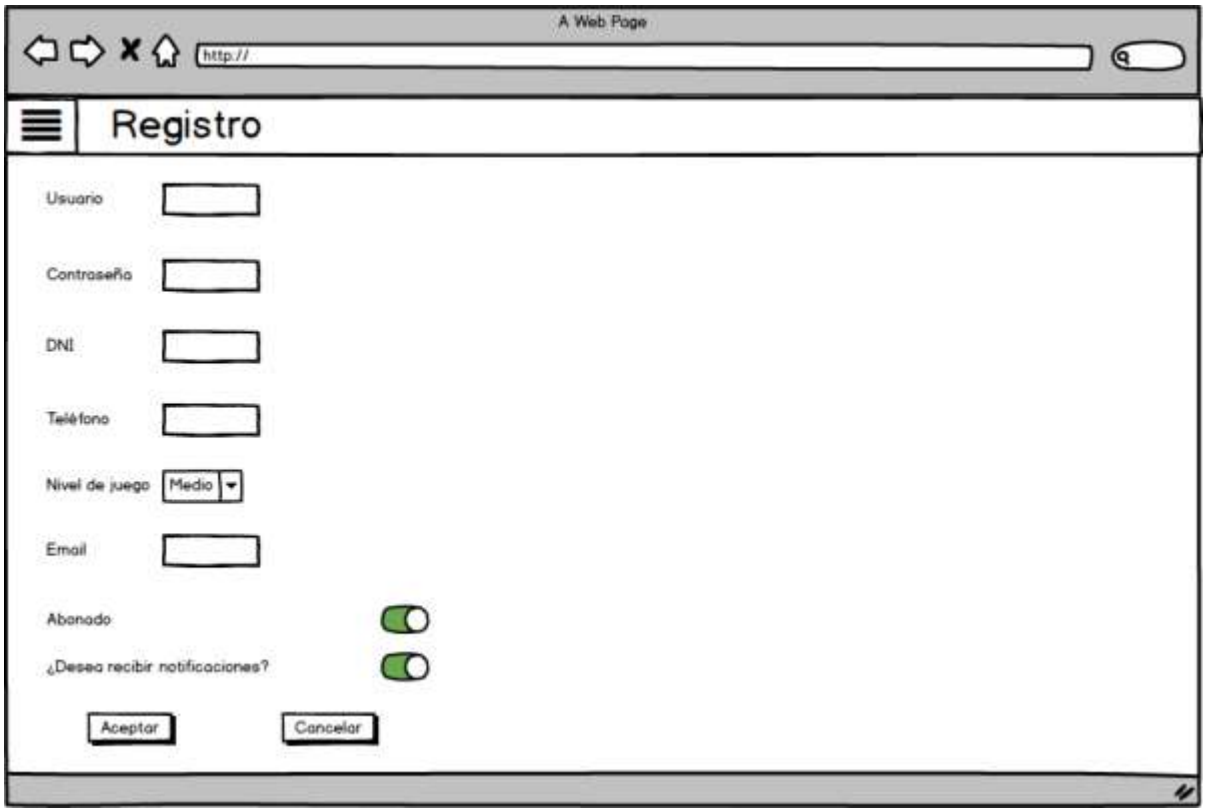
12.1. Prototipos de las interfaces de los usuarios registrados en la versión web

En este apartado se muestran los prototipos de las interfaces de la aplicación para *usuarios registrados* en su versión web.

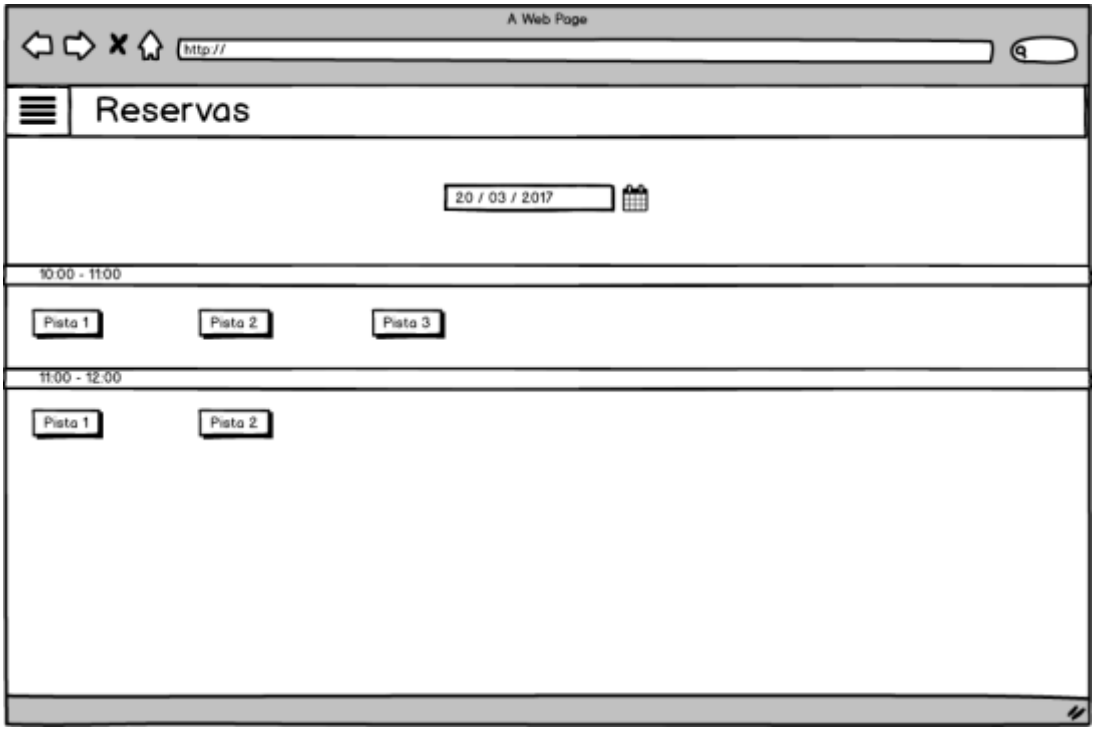
Pantalla de inicio de sesión para que los usuarios introduzcan su email y contraseña con la que se autentican en la aplicación:



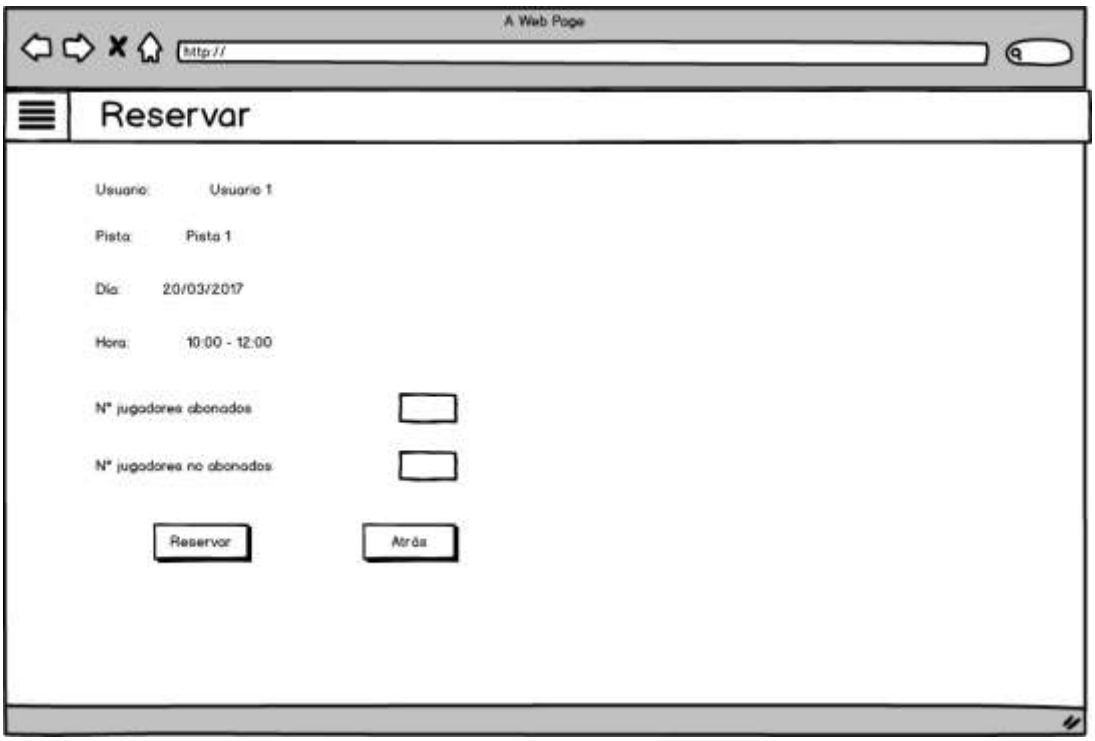
Pantalla de registro en la que los usuarios se dan de alta en la aplicación rellenando todos sus datos personales de forma correcta siendo el valor de *usuario* único para cada usuario:



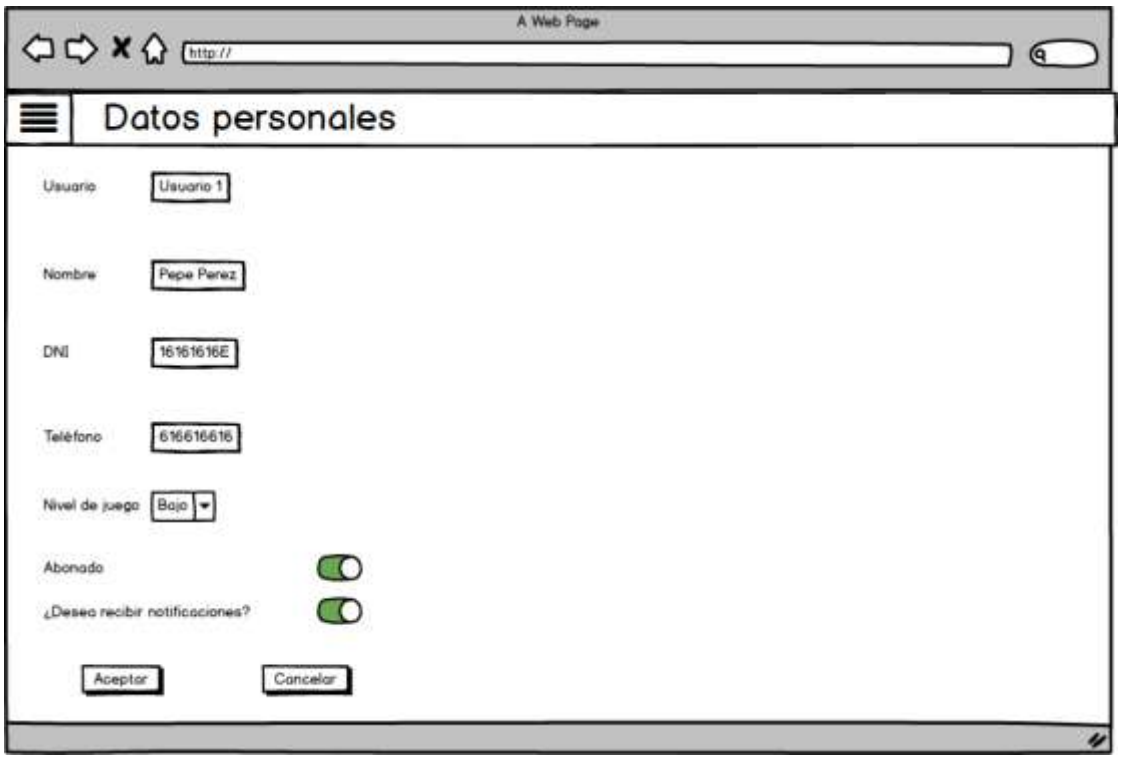
Pantalla para ver las pistas disponibles en los diferentes días y horas:



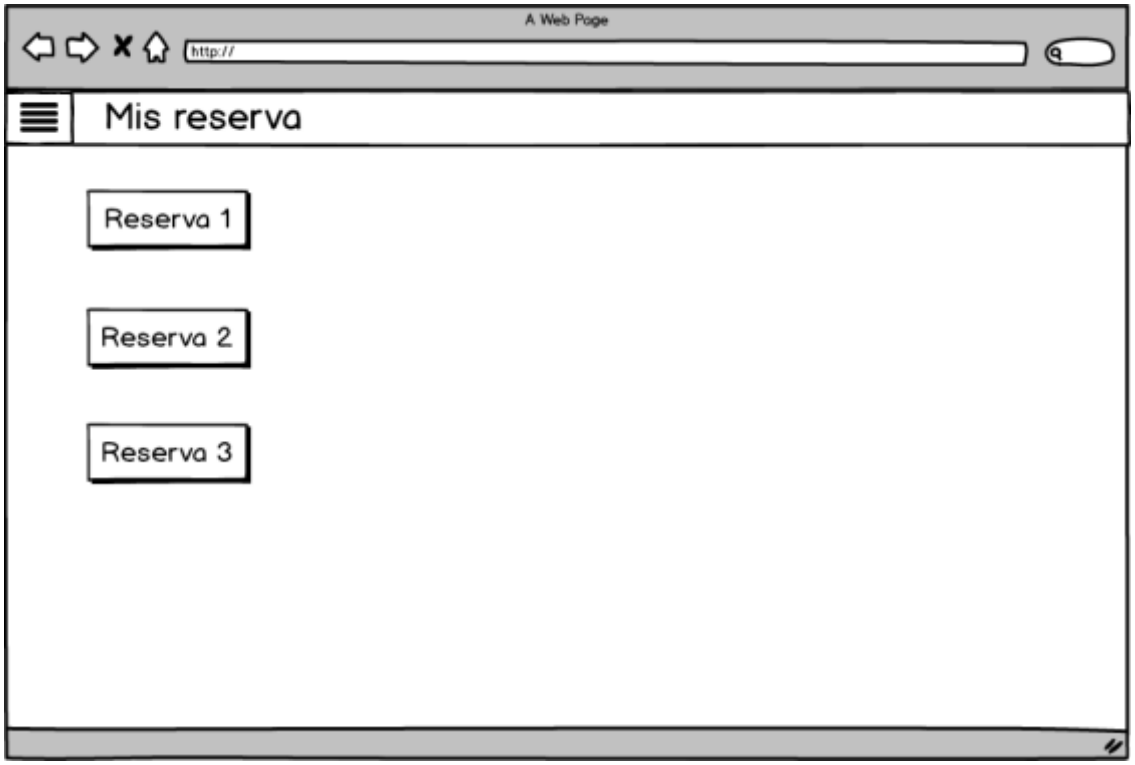
Pantalla para reservar una pista en la que el usuario solo podrá introducir en número de jugadores abonados y el de no abonados:



Pantalla donde el usuario puede modificar los datos personales siempre introduciendo valores válidos:



Y por último la pantalla para ver el historial de las reservas realizadas por el usuario en la aplicación donde podrá pinchar en cada una de ellas para ver los detalles de la reserva:



12.2. Preparación del entorno

En primer lugar se ha preparado el entorno en el que se va a programar la aplicación. Para ello se han seguido los diferentes pasos que se van a explicar a continuación. Antes de explicar los pasos a seguir decir que el sistema operativo usado en todo momento para el desarrollo del proyecto es *Windows*, cambiando entre *Windows 7* y *Windows 10*, debido a que el proyecto se ha realizado en la empresa Netbrain, en el que usábamos un ordenador con *Windows* 7, y también en el ordenador personal del alumno, que tiene un *Windows* 10. Se han seguido los mismos procedimientos en ambas versiones de *Windows*.

El primer paso a realizar es instalar *Ionic 2*. Para ello lo primero que hay que hacer es instalar *NodeJS* y *Cordova*. Una vez instalados, se procede a instalar *Ionic 2*.

Una vez instalado *Ionic 2*, creamos un proyecto de *Ionic 2*. Para ello, accedemos al directorio en el que se desea crear el proyecto con la consola de comandos y ejecutamos el siguiente comando para crear el proyecto:

```
ionic start nombre_proyecto sidemenu –v2
```

donde en `nombre_proyecto` ponemos el nombre que deseamos que tenga nuestro proyecto. La palabra `sidemenu` hace que el proyecto creado tenga generado ya por defecto la estructura para incluir el menú desplegable que hemos visto que deseamos que tengan las interfaces.

A continuación, tenemos que indicar las plataformas en las que queremos que funcione nuestra aplicación. Para conseguirlo, accedemos al directorio en el que se encuentra nuestro proyecto y ejecutamos el siguiente comando una vez por cada plataforma que queramos añadir a nuestro proyecto:

```
ionic platform add nombre_plataforma
```

Poniendo en `nombre_plataforma` el nombre de la plataforma que vamos a añadir.

Con los pasos anteriores ya tendríamos preparado el entorno de *Ionic 2*, pero es necesario añadir a este proyecto el sistema de base de datos que vamos a usar que es *Firebase*.

Para ello debemos crear una cuenta en *Firebase* e incluir un nuevo proyecto dentro de ésta. Cuando ya tengamos el proyecto creado en *Firebase*, tenemos que añadir este proyecto a nuestro proyecto en *Ionic 2*. Para ello, tenemos que acceder de nuevo a nuestro directorio del proyecto de *Ionic 2* con la consola de comandos de *Windows* y ejecutamos los siguientes comandos:

```
npm install @ionic/app-scripts@latest –save-dev
```

```
npm install firebase angularfire2 –save
```

De esta forma hemos importado las dependencias de *Firebase* desde nuestro proyecto en *Ionic 2*. El siguiente paso consiste en añadir el proyecto que hemos creado en *Firebase* a nuestro proyecto en *Ionic*. En nuestro caso, el código a añadir es el siguiente:

```
export const firebaseConfig = {
    apiKey: "AIzaSyDHaNLG_[redacted]",
    authDomain: "padel-56238.firebaseapp.com",
    databaseURL: "https://padel-56238.firebaseio.com",
    projectId: "padel-56238",
    storageBucket: "padel-56238.appspot.com",
    messagingSenderId: "172[redacted]"

};
```

12.3. Otros problemas encontrados

En esta sección se explican algunos problemas que hemos tenido a la hora de la implementación de la aplicación.

He tenido bastantes problemas con la interfaz de modificación de los datos personales de los usuarios. Este problema ha sido debido a que para la creación de formularios utilizaba un `<form>` en la vista, pero con la peculiaridad de añadirle que sea un `FormGroup` para que sea más fácil el paso de datos entre la vista y el controlador.

Por lo tanto, para usar este elemento, en la vista, a cada elemento del formulario hay que añadirle un atributo llamado `formControlName` como podemos ver en el ejemplo siguiente:

```
<ion-item>
    <ion-label >Nombre de usuario</ion-label>
    <ion-input type="text" id="usuario" formControlName="usuario" value="{{us.usuario}}"></ion-input>
</ion-item>
```

En el código anterior se puede ver la estructura de cada elemento que aparece en el formulario que como vemos es un `<ion-item>`. Dentro de este elemento, hay un elemento de tipo texto y otro elemento de entrada (de tipo texto) que va a recoger su valor del formulario. El valor por defecto del elemento de entrada viene definido por el controlador en su atributo `value`.

Cuando ya hemos añadido todos los elementos en la vista con su nombre asignado en el formulario (mediante el atributo `formControlName`), tenemos que crear una variable en el controlador del tipo `FormGroup`, que va a contener todos los elementos del formulario.

El código sería el siguiente:

```
this.datos = new FormGroup({
    nombre: new FormControl(this.nom),
    dni: new FormControl(this.dn),
    telefono: new FormControl(this.tel),
    nivel: new FormControl(this.ni),
    usuario: new FormControl(this.nomus),
    abonado: new FormControl(this.abonado)
})
```

Ahora utilizamos la función `value()` de esa variable para almacenar en otra variable todos los datos recogidos de la vista. Esta función sirve, por ejemplo, para cuando el usuario le dé al botón de `Guardar` en el caso de la interfaz para guardar los datos personales del usuario (ver página 21), podamos obtener los valores que ha introducido el usuario.

Hasta aquí todo funcionaba bien, pero el problema viene cuando hemos introducido un elemento en el formulario que no era una entrada de tipo texto ni de tipo numérico. Hemos introducido como podemos ver en los prototipos de las interfaces como la interfaz para ver los datos personales del usuario en las interfaces para *usuarios registrados* (ver página 21), un elemento que se llama `toggle` para que el usuario solo tuviera que pinchar sobre él para cambiar el valor de verdadero a falso. Este elemento lo he utilizado, inicialmente, para que el usuario indicase si quería notificaciones o no y si es abonado al club de pádel o no, pero como ya hemos dicho, las notificaciones no se han llevado a cabo y por lo tanto, finalmente, solo aparece este elemento para que el usuario indique si es abonado o no.

Este elemento, me ha dado muchos problemas porque al intentar recoger su valor al igual que el valor de todos los demás, no me devolvía el valor deseado y a veces hacía que dejase de funcionar la aplicación sin saber la razón. Después de mirar en la documentación y de buscar información por internet, decidí implementarlo de diferente forma.

Lo que hice es no incluir este elemento en el `FormGroup` y la creación de la variable para recoger los datos del formulario en el controlador quedó así:

```
this.datos = new FormGroup({
    nombre: new FormControl(this.nom),
    dni: new FormControl(this.dn),
    telefono: new FormControl(this.tel),
    nivel: new FormControl(this.ni),
    usuario: new FormControl(this.nomus)
})
```

Para conseguir el valor del `booleano` del elemento `toggle` lo obtuve de una de las formas que ya conocía (aprendido en la Universidad) que es utilizando el árbol “DOM” y obteniendo el elemento por su `id` de la vista de la siguiente forma:

```
let abonado = document.getElementById('abonado');
```

Ahora ya había conseguido obtener el valor del elemento, pero, a diferencia de los elementos del formulario anterior, al modificarlo el usuario, no se actualizaba su valor en el controlador y por lo tanto siempre me almacenaba en la base de datos el mismo valor. Para solucionar este problema, tuve que añadirle un evento a este elemento para que cuando el usuario pinchase sobre el elemento y su valor cambiase, llamase a una función que actualizase su valor. De esta forma, conseguía tener siempre actualizado el valor en el controlador para que al guardar los nuevos datos del usuario modificados, siempre guardase el valor actualizado. Además, conseguí que la aplicación ya no dejase de funcionar como lo hacía cuando introducía el `toggle` dentro del `FormGroup`. El resultado en la vista fue el siguiente:

```
<form [formGroup]="datos">
    <ion-list *ngFor="let us of usuario | async">
        <ion-item>
            <ion-label >Nombre de usuario</ion-label>
            <ion-input type="text" id="usuario" formControlName="usuario" value="{{us.usuario}}"></ion-i
        </ion-item>
        <ion-item>
            <ion-label >Nombre</ion-label>
            <ion-input type="text" id="nombre" formControlName="nombre" value="{{us.nombre}}"></ion-inpu
        </ion-item>
        <ion-item>
            <ion-label >DNI</ion-label>
            <ion-input type="text" id="dni" formControlName="dni" value="{{us.dni}}"></ion-input>
        </ion-item>
        <ion-item>
            <ion-label >Telefono</ion-label>
            <ion-input type="number" id="telefono" formControlName="telefono" value="{{us.telefono}}"></
        </ion-item>
        <ion-item>
            <ion-label >Nivel de juego</ion-label>
                <ion-select id="nivel" formControlName="nivel" required>
                    <ion-option>Bajo</ion-option>
                    <ion-option>Medio</ion-option>
                    <ion-option>Alto</ion-option>
                </ion-select>
        </ion-item>
        <ion-item>
            <ion-label >Abonado</ion-label>
            <ion-toggle id="abonado"  checked={{us.abonado}} (ionChange)="actualizarAbonado()"></ion-tog
        </ion-item>

    </ion-list>
```

Como vemos, el elemento `abonado` tiene un evento llamado `ionChange` que se ejecuta cada vez que este elemento es modificado y ejecuta la función `actualizarAbonado()` que modificar el valor de la variable del controlador que almacena el valor `booleano` que dice si el usuario es abonado o no lo es.